Advances for Resilient Real-Time Networks using Network Softwarization

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Steffen Philipp Lindner
aus Pforzheim

Tübingen
2023

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	06.02.2024
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Michael Menth
2. Berichterstatterin:	Prof. Dr.-Ing. Setareh Maghsudi
3. Berichterstatter:	Prof. Dr.-Ing. Georg Carle

Contents

List of Abbreviations

ABE	Alternative Best Effort
ALD	advanced loop detection
AQM	active queue management
BGP	Border Gateway Protocol
BE	best-effort
BIER	Bit Index Explicit Replication
BFIR	Bit-Forwarding Ingress Router
BFR	Bit-Forwarding Router
BFER	Bit-Forwarding Egress Router
CCDF	complementary cumulative distribution function
CUC	Centralized User Configuration
CLI	command line interface
DiffServ	differentiated services framework
DSCD	Deadlines, Saved Credits, and Decay
DSCP	DiffServ code point
eLFA	explicit LFA
FIFO	first-in-first-out
FRR	fast reroute
IAT	inter-arrival time
IETF	Internet Engineering Task Force
IPMC	IP multicast
L1	layer-1
L2	layer-2
L3	layer-3
LAA	Listener Attach Attribute
LFA	Loop-Free Alternate
MAT	match-action table
MPLS	Multiprotocol Label Switching
OF	OpenFlow
OSPF	Open Shortest Path First
P4	Programming Protocol-Independent Packet Processors
PSA	Portable Switch Architecture
PTE	protection tunnel egress
PTI	protection tunnel ingress
PNA	Portable NIC Architecture
QDisc	queuing discipline
QoS	Quality of Service

RAP	Resource Allocation Protocol
RTT	round trip time
RoLPS	Robust LFA Protection for Software-Defined Networks
rLFA	remote LFA
SDN	software-defined networking
TAA	Talker Announce Attribute
TAS	Time-Aware Shaper
TG	traffic generator
TI-LFA	topology-independent LFA
TSN	Time-Sensitive Networking
TNA	Tofino Native Architecture
VoIP	Voice over IP

Danksagung

Ein besonderer Dank gilt meinem Doktorvater Prof. Dr. Michael Menth, der mich in den vergangenen Jahren nicht nur fachlich, sondern auch menschlich gefördert und geprägt hat. Dank seiner herzlichen Art fühlten sich auch lange Papersessions, Wochenenden und scheinbar endlose Iterationen von Evaluation nicht nach Arbeit an. Neben seiner fachlichen Expertise war es insbesondere seine Persönlichkeit, die meine Freude an der Wissenschaft, dem Experimentieren und dem Entwickeln von neuen Lösungen geweckt hat. Einen so engagierten Doktorvater findet man selten. Vielen Dank!

Auch bei meiner Zweitgutachterin Prof. Dr.-Ing. Setareh Maghsudi und meinem Drittgutachter Prof. Dr.-Ing. Georg Carle möchte ich mich recht herzlich bedanken. Prof. Dr. Oliver Bringmann und Prof. Dr. Andreas Zell möchte ich für die Teilnahme als Prüfer bei meiner Disputation danken.

Meine Dissertation wäre nicht möglich gewesen, ohne die Zusammenarbeit mit meinen Kollegen und Studenten. Insbesondere möchte ich mich bei Daniel Merling, Marco Häberle, Lukas Osswald, Thomas Stüber, Gabriel Paradzik, Frederik Hauser, Benjamin Steinert, Jonas Primbs, Manuel Eppler, Moritz Flüchter, Fabian Ihle, Erfan Mozaffariahrar und Gülşen Ergün-Karagkiozidou bedanken. Ihr macht unseren Lehrstuhl zu dem, was er ist, und die tolle Atmosphäre lässt die Grenze zwischen Arbeit und Freizeit verschwimmen. Wir sind schließlich: Frollegen.

Zuletzt möchte ich mich bei meiner Familie, meinen Freunden, dem Wankheimer Italiener und allen, die an meiner Reise, die zu dieser Dissertation geführt hat, beteiligt waren, bedanken. Ein besonderer Dank gilt meinen Eltern, Andrea und Bernd, ohne deren Unterstützung diese Reise nicht möglich gewesen wäre. Mein abschließender Dank gilt meiner Freundin Patrizia, die mich immer unterstützt und an mich geglaubt hat.

Summary

Abstract

Traditional network devices, e.g., routers and switches, provide support for a fixed set of protocols and mechanisms. Network softwarization, and in particular software-defined networking (SDN) and programmable data planes, has emerged in recent years and breaks the previously existing vendor lock-in by decoupling the data and control plane of network devices. Thereby, the packet processing logic of network devices may be directly programmable, which enables the design of novel networking mechanisms without the manufacturer‘s direct support. As of today, Programming Protocol-Independent Packet Processors (P4) is one of the most common data plane programming technologies.

The research of this thesis focuses on two main objectives that aim to improve the state of the art for resilient real-time networks using network softwarization. The first objective is to develop and evaluate existing and novel resilient forwarding mechanisms using data plane programming. The second objective of this thesis is the support of real-time communication, i.e., the efficient support of BIER in P4, and concepts and algorithms for data transmission with Quality of Service (QoS) requirements. The main results of this thesis are published in five peer-reviewed publications. Seven additional peer-reviewed publications cover additional research results related to the main publications of this thesis, including a comprehensive P4 literature study.

The research presented in this thesis has been funded by different research projects by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/1-2, ME2727/2-1, and ME2727/3-1, the German Federal Ministry of Education and Research (BMBF) under support code 16KIS1161 (Collaborative Project KITOS), and the bwNET2020+ project, which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg (MWK).

Kurzfassung

Herkömmliche Netzwerkgeräte, wie beispielsweise Router und Switche, unterstützen einen während der Produktion festgelegten Umfang an Protokollen und Mechanismen. Network Softwarization, und insbesondere Software-Defined Networking (SDN) und Data Plane Programming, hat sich in den letzten Jahren immer stärker durchgesetzt und überwindet durch die Trennung von Data und Control Plane die zuvor bestehende Herstellerabhängigkeit. Dadurch kann die Paketverarbeitungslogik von kompatiblen Netzwerkgeräten direkt programmiert werden, was die Entwicklung neuartiger Netzwerkmechanismen ohne direkte Unterstützung durch den Hersteller ermöglicht. Heutzutage ist Programming Protocol-Independent Packet Processors (P4) eine der gängigsten Technologien zur Programmierung der Data Plane.

Die Forschung in dieser Arbeit konzentriert sich auf zwei Hauptziele, die darauf abzielen, den Stand der Technik für robuste Echtzeitnetzwerke mit Hilfe von Network Softwarization zu verbessern. Das erste Ziel ist die Entwicklung und Auswertung bestehender und neuartiger ausfallsicherer Weiterleitungsmechanismen unter Verwendung von Data Plane Programming. Das zweite Ziel dieser Arbeit ist die Unterstützung von Echtzeitkommunikation, d.h. die effiziente Unterstützung von BIER in P4, sowie Konzepte und Algorithmen zur Datenübertragung mit QoS Anforderungen. Die wichtigsten Ergebnisse dieser Arbeit sind in fünf peer-review Publikationen veröffentlicht. Sieben weitere peer-review Publikationen decken zusätzliche Forschungsergebnisse ab, die mit den Hauptpublikationen dieser Arbeit zusammenhängen, einschließlich einer umfassenden P4-Literaturstudie.

Die in dieser Arbeit vorgestellten Forschungsergebnisse wurden im Rahmen verschiedener Forschungsprojekte von der Deutschen Forschungsgemeinschaft (DFG) unter den Förderkennzeichen ME2727/1-2, ME2727/2-1 und ME2727/3-1, dem Bundesministerium für Bildung und Forschung (BMBF) unter dem Förderkennzeichen 16KIS1161 (Verbundprojekt KITOS) und dem Projekt bwNET2020+, das vom Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg (MWK) gefördert wird, finanziert.

List of Publications

The individual contributions to all publications (§ 6 Abs. 2 Satz 3 der Promotionsordnung) can be found in the appendix.

Accepted Manuscripts (Core Content)

1. Steffen Lindner, Daniel Merling, Marco Häberle, and Michael Menth. **P4-Protect: 1+1 Path Protection for P4** [LMHM20]. *P4 Workshop in Europe*, Barcelona, Spain, pp. 21-27, 2020. The author version of this publication can be found in the Appendix 1.1. The paper is also available online at the following URL: `https://doi.org/10.1145/3426744.3431327`

2. Daniel Merling, Steffen Lindner, and Michael Menth. **Hardware-Based Evaluation of Scalable and Resilient Multicast With BIER in P4** [MLM21a]. *IEEE Access*, vol. 9, pp. 34500-34514, 2021. The published version of this publication can be found in the Appendix 1.2. The paper is also available online at the following URL: `https://doi.org/10.1109/ACCESS.2021.3061763`

3. Steffen Lindner, Gabriel Paradzik, and Michael Menth. **Alternative Best Effort (ABE) for Service Differentiation: Trading Loss versus Delay** [LPM23]. *IEEE/ACM Transactions on Networking (ToN)*, vol. 31, no. 4, pp. 1642-1656, 2023. The author version of this publication can be found in the Appendix 1.3. The paper is also available online at the following URL: `https://doi.org/10.1109/TNET.2022.3221553`

4. Steffen Lindner, Daniel Merling, and Michael Menth. **Learning Multicast Patterns for Efficient BIER Forwarding with P4** [LMM23]. *IEEE Transactions on Network and Service Management (TNSM)*, vol. 20, pp. 1238-1253, 2023. The author version of this publication can be found in the Appendix 1.4. The paper is also available online at the following URL: `https://doi.org/10.1109/tnsm.2022.3233126`

5. Steffen Lindner, Marco Häberle, and Michael Menth. **P4TG: 1 Tb/s Traffic Generation for Ethernet/IP Networks** [LHM23]. *IEEE Access*, vol. 11, pp. 17525-17535, 2023. The author version of this publication can be found in the Appendix 1.5. The paper is also available online at the following URL: `https://doi.org/10.1109/ACCESS.2023.3246262`

Accepted Manuscripts (Additional Content)

6. Steffen Lindner, Marco Häberle, Florian Heimgärtner, Naresh Nayak, Sebastian Schildt, Dennis Grewe, Hans Loehr, and Michael Menth. **P4 In-Network Source Protection for Sensor Failover** [LHH$^+$20]. *International Workshop on Time-Sensitive and Deterministic Networking (TENSOR)*, Paris, France, pp. 791-796, 2020. The author version of this publication can be found in the Appendix 2.1. The paper is also available online at the following URL: `https://ieeexplore.ieee.org/abstract/document/9142735`

7. Daniel Merling, Steffen Lindner, and Michael Menth. **Comparison of Fast-Reroute Mechanisms for BIER-Based IP Multicast** [MLM20a]. *International Conference on Software Defined Systems (SDS)*, Paris, France, pp. 51-58, 2020. The author version of this publication can be found in the Appendix 2.2. The paper is also available online at the following URL: `https://doi.org/10.1109/SDS49854.2020.9143935`.

8. Daniel Merling, Steffen Lindner, and Michael Menth. **P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast** [MLM20b]. *Journal of Network and Computer Applications (JNCA)*, vol. 169, pp. 102764, 2020. The author version of this publication can be found in the Appendix 2.3. The paper is also available online at the following URL: `https://doi.org/10.1016/j.jnca.2020.102764`

9. Lukas Osswald, Steffen Lindner, Lukas Wuesteney, and Michael Menth. **RAP Extensions for the Hybrid Configuration Model** [OLWM21]. *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vasteras, Sweden, pp. 1-8, 2021. The author version of this publication can be found in the Appendix 2.4. The paper is also available online at the following URL: `https://doi.org/10.1109/ETFA45728.2021.9613246`

10. Daniel Merling, Steffen Lindner, and Michael Menth. **Robust LFA Protection for Software-Defined Networks (RoLPS)** [MLM21b]. *IEEE Transactions on Network and Service Management (TNSM)*, vol. 18, pp. 2570-2586, 2021. The author version of this publication can be found in the Appendix 2.5. The paper is also available online at the following URL: `https://doi.org/10.1109/TNSM.2021.3090843`

11. Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. **A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research** [HHM$^+$23]. *Journal of Network and Computer Applications (JNCA)*, vol. 212, 2023. The author version of this publication can be found in the Appendix 2.6. The paper is also available online at the following URL: `https://doi.org/10.1016/j.jnca.2022.103561`

12. Thomas Stüber, Lukas Osswald, Steffen Lindner, and Michael Menth. **A Survey of Scheduling Algorithms for the Time-Aware Shaper in Time-Sensitive Networking (TSN)** [SOLM23]. *IEEE Access*, vol. 11, pp. 61192-61233, 2023. The published version of this publication can be found in the Appendix 2.7. The paper is also available online at the following URL: `https://doi.org/10.1109/ACCESS.2023.3286370`

1 Introduction & Overview

Network softwarization, and in particular software-defined networking (SDN) and data plane programming, are highly relevant research topics in the field of communication networks. Traditional networking devices, e.g., routers and switches, provide support for a fixed set of protocols and mechanisms designated by the manufacturer. Although they may be highly configurable, they are most often only capable of providing new functionality with the manufacturer's direct support.

With SDN, network operators are able to extend the functionality of network devices to provide new services and features independently of the manufacturer. This may either be done through a well-defined interface between the forwarding and control plane or by making the devices themselves programmable.

In the following, two essential concepts, i.e., software-defined networking (SDN) and data plane programming with P4, are explained. Afterwards, the objectives of this thesis are described, and the main results are briefly summarized.

1.1 Software-Defined Networking

Software-defined networking (SDN) is widely considered to be the separation of data (also called forwarding) and control plane. The data plane is responsible for processing packets, whereas the control plane is responsible for its configuration. Figure 1.1 illustrates the difference between traditional networking, SDN with a fixed-function data plane, and SDN with a programmable data plane.

In traditional networking, both data and control plane are provided by the manufacturer and are tightly coupled. The control plane may offer configuration capabilities, e.g., through a web interface, command line interfaces (CLIs), or other means. However, both the functionality of the data plane as well as the control plane algorithms cannot

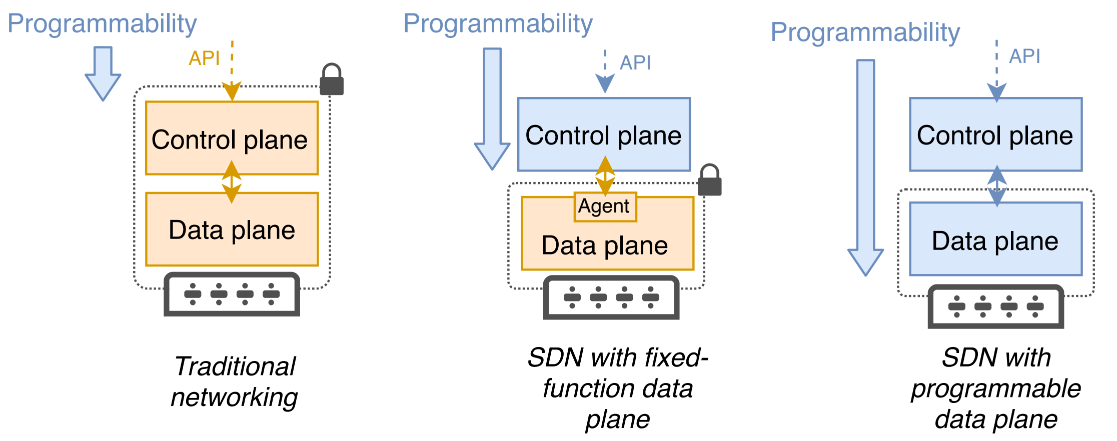


Figure 1.1: Differences between traditional networking, SDN with fixed-function data plane, and SDN with programmable data plane. Illustration from Hauser et al. [HHM+23].

be changed or replaced. Consequently, protocols and functions not envisioned by the manufacturer cannot be supported.

With SDN, the data and control plane are decoupled. There are two types of SDN, SDN with a fixed-function data plane and SDN with a programmable data plane. SDN with a fixed-function data plane provides a well-defined interface between the data and control plane that allows the configuration of the provided data plane, e.g., through the population of so-called match action entries for forwarding tables. This separation also allows complex decentralized algorithms, such as routing algorithms, to be replaced by a centralized control plane configuring multiple network devices. A prominent example of an interface between the data and control plane is OpenFlow (OF) [MAB+08]. OF-capable switches and routers provide a fixed-function data plane that can be configured through a user-provided control plane[1] with the help of the OF protocol.

In addition, SDN with programmable data planes does not only allow for a user-provided control plane but also a user-provided data plane. Thereby, the packet processing pipeline of network devices is described by software which allows the definition of new protocols and forwarding mechanisms. As of today, Programming Protocol-Independent Packet Processors (P4) [BDG+14] is one of the most common data plane programming technologies.

This fundamental property, i.e., that end users can program both the data and control

[1]Such a control plane may be written in any high-level programming language, e.g., Python or C++.

plane of network devices, breaks the previously existing manufacturer lock-in. As a result, development cycles for new products or concepts can be significantly reduced, as it is not necessary to first develop specialized hardware. Instead, existing hardware can be programmed according to the users' requirements.

1.2 Programming Protocol-Independent Packet Processors (P4)

Programming Protocol-Independent Packet Processors (P4) is a domain-specific programming language that is used to describe the data plane of P4-capable network devices. With [HHM+23], we published a comprehensive literature survey which is also part of this thesis that covers 519 research papers about P4 or leveraging P4. Further, it provides an introduction to the P4 ecosystem, a P4 tutorial, and summarizes hundreds of research papers. $P4_{16}$[2] is the most recent version of P4.

A P4 program follows a so-called P4 architecture that defines the available language constructs and the used processing pipeline. Examples of P4 architectures are the Portable Switch Architecture (PSA), Portable NIC Architecture (PNA), or Tofino Native Architecture (TNA). Devices executing P4 programs are called targets. P4 programs are translated through target-specific compilers into binaries for the corresponding P4 target. Figure 1.2 illustrates a simplified P4 processing pipeline that resembles the processing pipeline of most P4 architectures.

Packets are received on so-called ingress ports and afterwards parsed by a user-programmable packet parser. Thereby, packet headers are extracted, e.g., Ethernet and IPv4. The extracted headers are subsequently used in (possibly multiple) match-action tables (MATs) together with packet-specific metadata to decide which actions should be executed. Examples of user-defined actions are replacing certain header fields, forwarding packets through specific egress ports, or updating the contents of register[3] fields. More complex operations may be available as so-called externs[4] depending on the used P4 architecture. The mapping between header and metadata fields and the corresponding action is typically provided by the control plane. Finally, packets are serialized to the wire by a deparser and sent through a previously specified egress port.

[2]`https://p4.org/specs/`

[3]Registers are not part of the core P4 language and not available in all P4 architectures.

[4]An extern is an operation that is not defined in the core P4 language but in the architecture. Targets may provide almost arbitrary functions as externs, e.g., complex math operations.

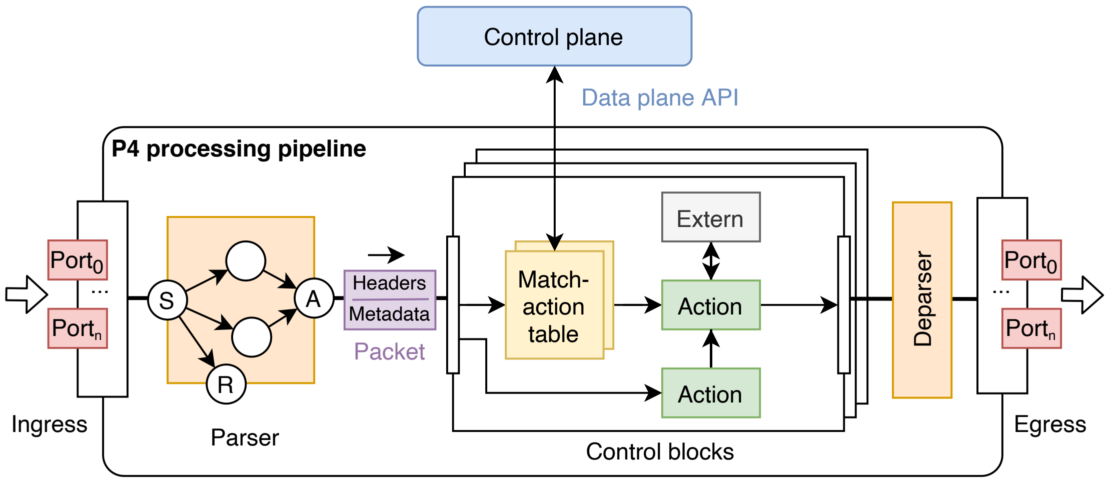


Figure 1.2: Simplified P4 processing pipeline. Illustration from Hauser et al. [HHM$^+$23].

The combination of a well-defined processing pipeline and programmable building blocks allows the design of novel, high-performance network solutions that range from legacy protocols such as Multiprotocol Label Switching (MPLS) to in-network acceleration of machine learning algorithms [HHM$^+$23].

1.3 Research Objectives

Communication networks are prone to service disruption, e.g., through network equipment failure. One possible solution to this problem is to leverage resilient forwarding mechanisms that may automatically reroute affected traffic through working backup paths to restore connectivity. This is especially relevant for real-time communication that cannot rely on the automatic retransmission of transport protocols. This thesis focuses on two main objectives that aim to improve the state of the art for resilience real-time networks using network softwarization, particularly with the help of SDN and programmable data planes.

The first objective is to develop and evaluate both existing and novel **resilient forwarding mechanisms using data plane programming**. This means that traffic affected by forwarding failures, e.g., caused by the failure of a communication link, should be forwarded by other means to restore connectivity as fast as possible, thereby minimizing the experienced packet loss.

The second objective of this thesis is the **support of real-time communication**. This objective is separated into two categories. First, the efficient support of BIER in P4. Bit Index Explicit Replication (BIER) is a novel transport mechanism for multicast traffic standardized by the Internet Engineering Task Force (IETF) [WRD$^+$17]. Multicast is often used for real-time applications such as financial stock exchange or IPTV [NAC$^+$20]. The second category comprises concepts and algorithms for data transmission with QoS requirements. Examples for QoS requirements are an upper bound for the end-to-end delay, a guaranteed bandwidth, or zero packet loss.

1.4 Research Context

The research presented in this thesis has been funded by different research projects by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/1-2, ME2727/2-1, and ME2727/3-1. Further, some work has been funded by the German Federal Ministry of Education and Research (BMBF) under support code 16KIS1161 (Collaborative Project KITOS) and the bwNET2020+ project, which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg (MWK). The published versions of the publications in the appendix indicate which work has been funded by which research project.

All research was carried out in collaboration with colleagues. A description of the contributions that my coworkers and I made to the individual works can be found in the appendix.

1.5 Research Results

This thesis comprises 12 publications. All publications can be found in the appendix. Chapter 2 summarizes and presents the research results of these publications. For each publication, the research objective and research results are presented. In the following, an overview of the research and its results is given.

One contribution of this thesis is a broad P4 literature study [HHM$^+$23] that reviews 519 research papers about P4. Several of the presented works in this thesis are directly influenced by the results of this extensive literature study that identified open research potential. One key insight of [HHM$^+$23] was that most work in the field of data plane programming is conducted with software-based switches such as the BMv2 [p4l19]. Software-based switches do not impose restrictions on the complexity of the algorithms as all processing is done entirely in software. Hardware-based switches on the other hand have only limited capabilities to ensure line rate processing. Consequently, it is unclear whether a large portion of the existing research in the field of data plane programming is applicable to hardware-based switches and thus applicable in reality. Therefore, great care has been taken to ensure that all presented algorithms and mechanisms that leverage data plane programming in P4 can also be implemented on hardware-based switches.

The first main objective of this thesis, i.e., **resilient forwarding mechanisms using data plane programming**, is covered in Section 2.1. One significant contribution is the development of a reliable 1+1 protection mechanism for IP networks in P4 [LHH+20] which is presented in Section 2.1.2. Thereby, traffic is forwarded over two disjoint paths such that any failure of a single path can be compensated. Further, the proposed mechanism may be used across the Internet. Evaluations show its feasibility in a 100 Gb/s environment. The second contribution is the development of a fast 1:1 protection scheme [MLM21b] based on Loop-Free Alternates (LFAs) [AZ08] in Section 2.1.3. Evaluations show that it protects against any single component failure and most double failures and runs at line rate in a 100 Gb/s environment. Finally, a P4-based mechanism for in-network detection of sensor failures [LHH+20] is presented in Section 2.1.4.

The second main objective of this thesis, i.e., **support of real-time communication**, is covered in Section 2.2 and split into two parts. The first part is the efficient support of BIER in P4 and comprises four publications. One contribution is the development of novel BIER fast reroute (FRR) mechanisms [MLM20a] in Section 2.2.1.2 and the implementation of BIER for the software-switch BMv2 in P4 [MLM20b] in Section 2.2.1.3. The second contribution is the implementation of BIER and BIER-FRR [MLM21a] for the Intel Tofino™, a high-performance P4 switching ASIC, in Section 2.2.1.4. Extensive evaluations show that the BIER implementation runs at 100 Gb/s line rate, but also reveal processing limits caused by excessive recirculation overhead. The processing overhead caused by excessive recirculation is addressed in Section 2.2.1.5, where a novel BIER processing scheme for the Intel Tofino™ is introduced [LMM23]. Simulations and hardware evaluations show that the new processing scheme significantly decreases the processing overhead for BIER packets. The proposed mechanism is further optimized through the help of clustering, an unsupervised machine-learning technique.

One significant contribution for the second part of the second main objective, i.e., data transmission with QoS requirements, is the design and implementation of a novel packet scheduler, called DSCD [LPM23], in Section 2.2.2.2. DSCD is implemented in the Linux network stack and enables low-delay forwarding for real-time traffic without degrading the performance of common best-effort (BE) traffic. Extensive evaluations show its feasibility and analyze its properties. Other contributions are the analysis and extension of the RAP protocol for Time-Sensitive Networking (TSN) networks in Section 2.2.2.3 and a survey reviewing the existing scheduling literature for the Time-Aware Shaper (TAS) in TSN networks. The last contribution of this thesis is the design and implementation of a low-cost traffic generator (TG), called P4TG [LHM23], based on the Intel Tofino™ in Section 2.2.2.5. It is capable of generating up to 1 Tb/s traffic split across $10\times$ 100 Gb/s ports.

2 Results & Discussion

This chapter summarizes and discusses the results of this thesis. Section 2.1 contains research on the first objective, i.e., resilient forwarding mechanisms using programmable data planes. Section 2.2 presents the second objective, i.e., support of real-time communication. To that end, Section 2.2.1 covers efficient support of BIER in P4, and Section 2.2.2 discusses data transmission with QoS requirements.

For each presented work, it is indicated whether they are part of the core content of this thesis or part of the additional content.

2.1 Resilient Forwarding Mechanisms using Data Plane Programming

This section summarizes the research results of this thesis on resilient forwarding mechanisms using data plane programming. It comprises three publications. One publication is part of the core content of this thesis, and two are part of the additional content. First, foundations for resilient forwarding mechanisms are introduced. Then, the three publications are summarized.

2.1.1 Foundations

Routing algorithms, such as the Border Gateway Protocol (BGP) [RHL06] or the Open Shortest Path First (OSPF) protocol [Moy98], determine the default forwarding behavior for data packets. Data packets may be lost when a communication link fails or when an intermediate forwarding node (next-hop) is not reachable anymore. Affected traffic can only reach its destination until the default forwarding paths have been recalculated and updated on the forwarding devices, which is called reconvergence. However, this procedure may take up to several hundreds of milliseconds [LPS$^+$13].

Legacy networks circumvent this problem by introducing fast-recovery mechanisms, such as FRR or 1+1 protection. With FRR, packets are rerouted leveraging precomputed backup paths without the need for reconvergence or any other control plane interaction. With 1+1 protection, traffic is simultaneously sent through two disjoint paths towards the destination. If one path is affected by a failure, the second path ensures that the data packets are still received. Chiesa et al. [CKR^{+}21] give a broad overview of fast-recovery mechanism in packet-switched networks. Further details on different FRR mechanisms and related surveys can be found in [MLM21b].

2.1.2 P4-Protect: 1+1 Path Protection for P4

This section summarizes and presents the research results from Lindner et al. [LMHM20]. This publication is part of the core content of this thesis. Further details can be found in the associated publication. First, the research objective is introduced. Then, the solution and evaluation results are explained.

2.1.2.1 Research Objective

FRR methods such as LFAs [AZ08] send traffic to an alternative next-hop when the primary next-hops is not reachable anymore. However, failure detection and rerouting are not instantaneous. Therefore, packet loss cannot be avoided. With 1+1 protection mechanisms, traffic is duplicated and sent over two disjoint paths and deduplicated by the receiver. When one of the disjoint paths is affected by a failure, the second path still ensures that the traffic is correctly forwarded.

The goal of this work was to develop a 1+1 protection mechanism for IP networks that may even be useable across foreign networks.

2.1.2.2 1+1 Protection Mechanism

P4-Protect is a 1+1 protection scheme based on P4 [LMHM20]. To that end, traffic is duplicated at a duplication node, forwarded through two disjoint paths across a network, and deduplicated by a deduplication node. The deduplication node only forwards the first version of each packet. Figure 2.1 illustrates the concept of P4-Protect.

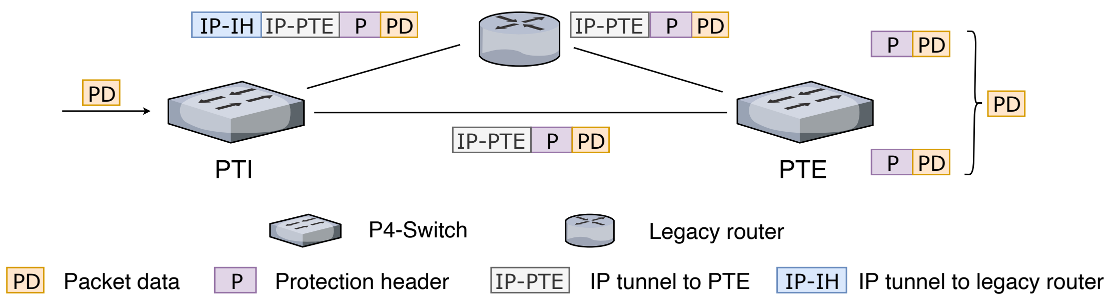


Figure 2.1: Concept of P4-Protect. Illustration from [LMHM20].

A so-called protection tunnel ingress (PTI) receives packets and applies the protection procedure. It adds a new header, the so-called protection header, to the packet which includes a sequence number and a connection identifier. The packet is then tunneled via two preferably disjoint paths to the so-called protection tunnel egress (PTE) which terminates the protected connection and deduplicates the traffic. A unique feature of P4-Protect is that it can be used over the Internet where not all intermediate nodes are under the control of the same network operator. With 1+1 protection over the Internet, one problem is that it is not possible to ensure that the used paths are disjoint. To that end, P4-Protect allows tunneling the traffic on one path to an additional intermediate node. There, the packet will be decapsulated and forwarded to the PTE. Thus, the intermediate node can enforce some path disjointness.

2.1.2.3 Implementation in P4

We implemented P4-Protect for the software switch BMv2 [p4l19] and the Intel Tofino™ ASIC, a high-performance switching ASIC. In this work, we leveraged an Edgecore Wedge 100BF-32X [Edg21] switch based on Intel Tofino™ 1 with 32 100 Gb/s ports. One particular challenge was implementing the deduplication procedure on the Intel Tofino™, as hardware targets have certain limitations on the number of operations that can be applied per packet to ensure line rate processing.

2.1.2.3.1 Deduplication Procedure When the PTE receives a packet, it has to decide whether it is forwarded or dropped because it has already been received. To that end, the PTE stores the last accepted sequence number SN_{last}^{PTE}. Further, it leverages a so-called acceptance window W to accept sequence numbers in a given range. This is required if the same packet is lost on both paths. As the sequence number space is not unlimited, i.e., it is cyclic, a sequence number SN larger than SN_{last}^{PTE} may indicate a new packet, but it may also result from a very old packet. The equations to ensure that an incoming packet falls within the acceptance window can be easily derived. Let SN_{max} be the maximum sequence number. If $SN_{last}^{PTE} + W < SN_{max}$ holds, a packet with sequence number SN is accepted if

$$SN_{last}^{PTE} < SN \leq SN_{last}^{PTE} + W \tag{2.1}$$

If $SN_{last}^{PTE} + W \geq SN_{max}$ holds[1], a packet with sequence number SN is accepted if one of the following inequalities holds:

$$SN_{last}^{PTE} < SN \tag{2.2}$$

$$SN < SN_{last}^{PTE} + W - SN_{max} \tag{2.3}$$

A packet copy can arrive $SN_{max} - W$ sequence numbers after the first copy without being recognized as a new packet. SN_{last}^{PTE} is set to SN if the packet is accepted.

Due to hardware limitations, the presented check cannot be implemented. Therefore, we set $W = \frac{SN_{max}}{2}$ to simplify the required inequalities. If $W \leq SN$ holds, both inequalities must be true:

$$SN_{last}^{PTE} < SN \tag{2.4}$$

$$SN - SN_{last}^{PTE} \leq W \tag{2.5}$$

If $SN < W$, only one of the inequalities has to hold:

$$SN_{last}^{PTE} < SN \tag{2.6}$$

$$W \leq SN_{last}^{PTE} - SN \tag{2.7}$$

2.1.2.4 Evaluations

In this section, we evaluate P4-Protect. First, we consider the achieved TCP goodput. Then, we assess its impact on packet jitter.

[1]This means that an overflow of the sequence number range has to be considered.

2.1.2.4.1 TCP Goodput We evaluate the achievable TCP goodput with P4-Protect on the Edgecore Wedge 100BF-32X [Edg21]. To that end, we set up iperf3 [ipe] connections between multiple client/server pairs connected through two disjoint paths and measure their goodput. Each connection consists of 15 parallel TCP flows. Figure 2.2 shows the TCP goodput with P4-Protect with a varying number of client/server pairs exchanging traffic with iperf3.

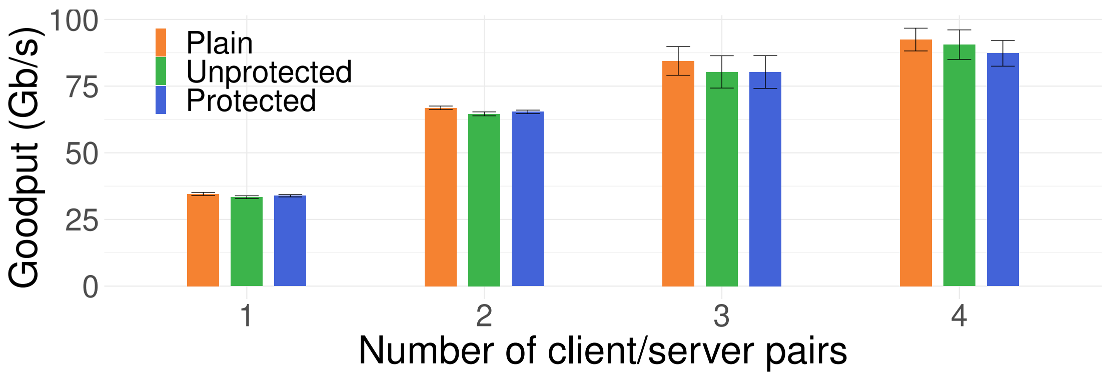


Figure 2.2: TCP goodput with P4-Protect with a varying number of client/server pairs exchanging traffic with iperf3. Figure from [LMHM20].

The experiment was performed 20 times, and the 95% confidence intervals are provided. Three forwarding modes are considered, i.e., plain, unprotected, and protected. Plain forwarding is a simplified IP forwarding plane without any protection that leverages only a single path. Unprotected is the P4-Protect program without activated protection, i.e., traffic is not duplicated. Protected is P4-Protect with activated protection, i.e., traffic is duplicated, sent via disjoint paths, and deduplicated. Less than four client/server pairs cannot saturate a 100 Gb/s connection which is expected with default TCP traffic. With four client/server pairs, a goodput of around 90 Gb/s is achieved. This is less than 100% because the overhead of Ethernet, IP, and TCP is not part of the goodput. All three forwarding modes lead to almost identical goodput. The goodput for protected forwarding is slightly less than unprotected forwarding due to the header overhead of P4-Protect.

2.1.2.4.2 Impact on Jitter The PTE forwards the first copy of each packet. To that end, jitter on the individual paths may be smoothed. We verify this assumption through the following experiment. Two hosts are connected through two disjoint paths with an artificial, adjustable, uniformly distributed jitter. We send pings between the hosts and measure the average round trip time (RTT). Figure 2.3 shows the impact of P4-Protect on packet jitter.

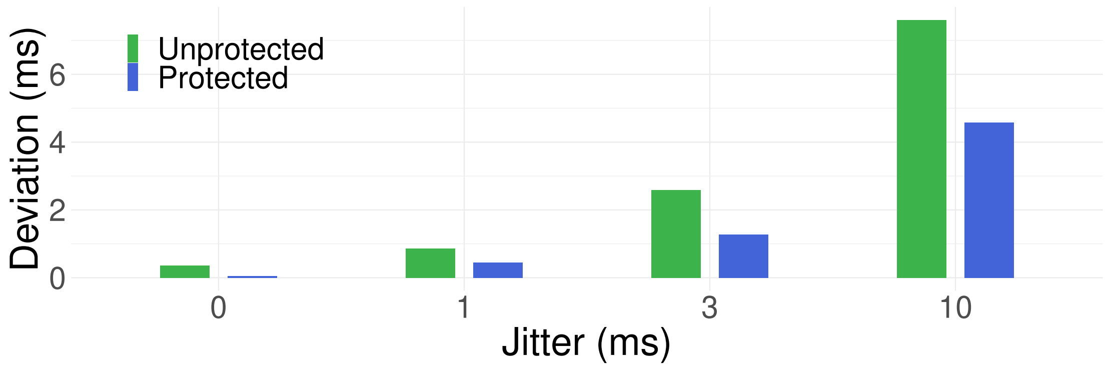


Figure 2.3: Impact of P4-Protect on packet jitter. Figure from [LMHM20].

With unprotected traffic, the average RTT deviation corresponds to the configured average jitter on the path. With P4-Protect, protected traffic suffers only about half the configured jitter. Therefore, the PTE smoothes the experienced end-to-end jitter as it forwards the first version of each packet.

2.1.2.5 Conclusion & Discussion

With P4-Protect, we designed a 1+1 protection scheme that runs at high data rates, smoothes experienced end-to-end jitter, and effectively protects against the failure of individual paths. Further, it may be suited to be used across the Internet.

We published and presented P4-Protect [LMHM20] on the P4 Workshop in Europe (EuroP4) in 2020 (see Appendix 1.1). Further, P4-Protect has been published as an open-source implementation on Github[2].

2.1.3 Robust LFA Protection for Software-Defined Networks (RoLPS)

This section summarizes the research results from Merling et al. [MLM21b]. This publication is part of the additional content of this thesis. Therefore, it is only briefly summarized. Further details can be found in the associated publication. First, the research objective is introduced. Then, the work is summarized.

[2]`https://github.com/uni-tue-kn/p4-protect-tofino`

2.1.3.1 Research Objective

1+1 protection mechanisms like P4-Protect have the advantage that almost any packet loss can be prevented. However, they also require at least twice the bandwidth as traffic is replicated to two paths. Alternatively, there are 1:1 mechanisms. With 1:1 mechanisms, traffic is forwarded through a primary path. Once an error is detected, the traffic is transmitted along an alternative path.

In earlier work, Merling et al. [MBM18] augment the concept of LFAs [AZ08]. They introduced so-called explicit LFA (eLFA) that leverage explicit tunnels to protect destinations against failures that cannot be protected by other LFAs. Further, an advanced loop detection (ALD) mechanism has been proposed to detect and stop routing loops that may occur in the case of multiple network failures.

The goal of this work was to augment the existing eLFAs and improve the ALD mechanism so that it is implementable on P4 devices.

2.1.3.2 Summary

RoLPS augments the previously proposed eLFAs through multipoint-to-point rerouting tunnels. Thereby, multiple eLFAs may share state in forwarding devices, such that the overall required forwarding state is significantly decreased. Previously conducted simulations regarding the coverage of different LFAs variants are updated and extended with topology-independent LFAs (TI-LFAs). The results show that existing LFAs and so-called remote LFAs (rLFAs) cannot protect all destinations against failures and that forwarding loops may occur in the case of multiple network failures. Only eLFAs and TI-LFAs are able to protect all destinations. Subsequently, an implementation of an improved ALD mechanism as well as an implementation of LFAs, rLFAs, eLFAs, and TI-LFAs for the Intel Tofino™ are presented. Hardware evaluations reveal that affected traffic is successfully rerouted after a short restoration time of 0.6 ms, that routing loops are detected and stopped, and that the implementation runs at 100 Gb/s per port.

We published RoLPS as a journal paper in the Special Section on Design and Management of Reliable Communication Networks of IEEE Transactions on Network and Service Management (TNSM) [MLM21b] (see Appendix 2.5) in 2021. The source code of the Tofino-based prototype has been published as an open-source implementation on Github[3].

2.1.4 P4 In-Network Source Protection for Sensor Failover

This section summarizes and presents the research results from Lindner et al. [LHH$^+$20]. This publication is part of the additional content of this thesis. Further details can be found in the associated publication. First, the research objective is introduced. Then, the solution and evaluation results are explained.

2.1.4.1 Research Objective

Systems in industrial settings, e.g., factory automation or autonomous driving, primarily depend on information obtained from sensors. Therefore, several sensors may provide the same data to an application to prevent service disruption in the case of a sensor failure. The receiving application may either use both data streams or rely on a single data source until the primary stream fails, which increases the complexity of the application and is prone to errors. The goal of this work was to develop an in-network sensor failover mechanism that detects the failure of a primary sensor and delivers in turn the data of a redundant sensor to the application.

2.1.4.2 Summary

In this work, we proposed two mechanisms to detect the failure of a periodic primary sensor directly in the network. Upon failure detection, the data of a redundant sensor is forwarded. The first mechanism is based on a counter-based approach. For each arriving data portion from the redundant sensor, a counter is increased. In contrast, for each arriving data portion of the primary sensor, the counter is set to zero. If the counter exceeds a threshold t, the data of the redundant sensor is forwarded. The second mechanism leverages a timer-based approach and improves reliability in the case of unstable sensor periods. With the timer-based approach, the timestamp of the last received primary sensor data portion is stored. Data from the redundant sensor is forwarded if

[3]`https://github.com/uni-tue-kn/p4-lfa`

the elapsed time since the last data from the primary sensor exceeds a given threshold. The counter-based approach was implemented for the Intel Tofino™ whereas the timer-based approach was implemented for the software switch BMv2.

This work [LHH+20] has been published and presented at the International Workshop on Time-Sensitive and Deterministic Networking (TENSOR) in 2020 (see Appendix 2.1).

2.2 Support of Real-Time Communication

This section summarizes the research results of this thesis regarding the support of real-time communication. This objective is separated into two categories. First, the efficient support of BIER in P4. The second category comprises concepts and algorithms for data transmission with QoS requirements.

2.2.1 Efficient Support of BIER in P4

This section summarizes the research results of this thesis on the efficient support of BIER in P4. It comprises four publications. Two publications are part of the core content of this thesis, and two are part of the additional content. First, an overview of BIER is given. Then, the four publications are summarized. The first publication investigates and compares FRR methods for BIER. The second publication presents an implementation of BIER and BIER-FRR in P4 for the software switch BMv2. In the third publication, we implemented BIER and BIER-FRR on the hardware target Intel Tofino™ and performed extensive evaluations. Finally, the fourth publication dramatically increased the efficiency of the BIER forwarding through the help of switch-intern static multicast groups and an optimization based on machine learning.

2.2.1.1 BIER Overview

Traditional IP multicast (IPMC) is used when a sender wants to transmit the same packet to multiple receivers. Examples of use cases are live-streaming, IPTV or financial stock exchange [NAC+20]. Receivers can join so-called multicast groups, and the sender addresses this multicast group as the packet recipient. The information which network nodes require a packet copy for a given multicast group is propagated through the network with the help of multicast routing protocols. Thus, each node in the network must store which neighbor needs a packet copy for which multicast group. On the one hand, this requires a lot of memory. On the other hand, as soon as a multicast group changes, e.g., because a new receiver has joined or another receiver has left the group, this information has to be propagated again through the entire network.

Bit Index Explicit Replication (BIER) [WRD$^+$17] has been proposed by the Internet Engineering Task Force (IETF) to solve the problems of traditional IPMC. BIER introduces a domain concept where only ingress border routers, called Bit-Forwarding Ingress Routers (BFIRs), need to know which receivers require a packet copy for a given multicast group. Figure 2.4 illustrates the concept of BIER.

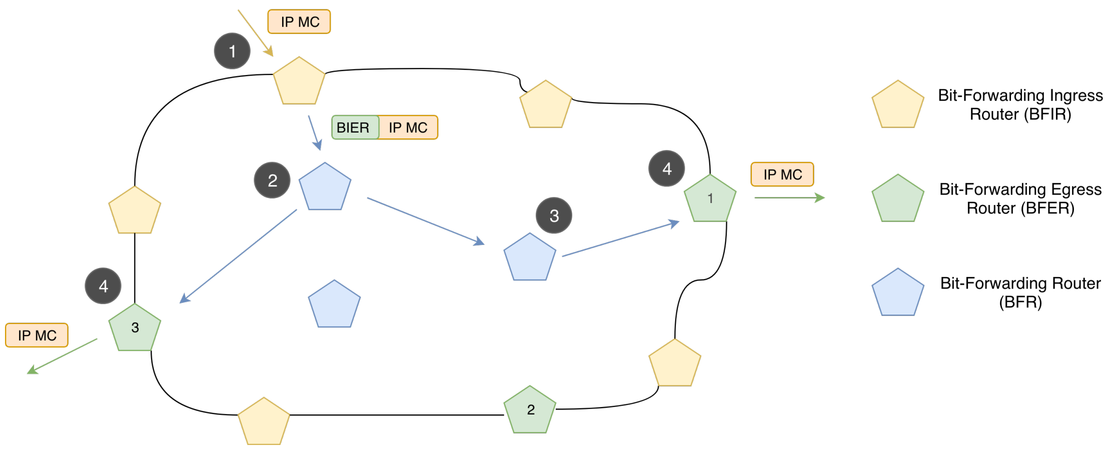

IP MC
1
BIER
IP MC
2
3
4
1
IP MC
4
3
IP MC
2
Bit-Forwarding Ingress Router (BFIR)
Bit-Forwarding Egress Router (BFER)
Bit-Forwarding Router (BFR)

Figure 2.4: A BIER domain consists of a Bit-Forwarding Router (BFR), Bit-Forwarding Ingress Router (BFIR), and Bit-Forwarding Egress Router (BFER). Illustration from [Lin19] adapted from [MMWE18].

The original IPMC packet is encapsulated with a BIER header containing a so-called BIER bitstring ❶. Each bit in the BIER bitstring corresponds to an egress border router, called BFER, of the BIER domain. If the bit is activated, i.e., set to 1, it indicates that the corresponding BFER requires a packet copy. This information is leveraged by forwarding nodes of the BIER domain, called BFR. When a BFR receives a BIER packet, it replicates and forwards it towards all destinations indicated by the BIER bitstring ❷ ❸. Thereby, bits of BFERs that are reached via other neighboring BFRs are cleared in the bitstring to prevent duplicates at the receiver. Forwarding takes place along the paths of the routing underlay, e.g., paths computed by OSPF. Consequently, a BFR only needs to know which BFER is reachable through which neighbor, which is independent of the actual multicast group. Therefore, even when multicast groups change, there is no need to update the forwarding state on BFRs. Only the BFIRs must be updated when multicast groups change. Finally, when a BIER packet reaches a BFER, the BIER header is removed and the underlying IPMC packet is forwarded as usual ❹.

2.2.1.2 Comparison of Fast-Reroute Mechanisms for BIER-Based IP Multicast

This section summarizes the research results from Merling et al. [MLM20a]. This publication is part of the additional content of this thesis. Therefore, it is only briefly summarized. Further details can be found in the associated publication.

With [MLM20a] we developed and compared two approaches to protect BIER traffic with FRR mechanisms. The first mechanism is based on LFAs, similar to one of the presented mechanisms of RoLPS (see Section 2.1.3). When a neighboring BFR is not reachable, the BIER packet is forwarded to an alternative next-hop and the BIER bitstring is adapted accordingly. The second mechanism is based on the protection properties of the unicast routing underlay, e.g., IP-FRR. When a BFR is not reachable, BIER packets are encapsulated with unicast tunnels towards the next-hop or next-next-hop. Then, the routing underlay may use its own protection mechanisms to reroute the packet to its destination, e.g., with LFAs. When the neighboring BFR is reached, the unicast tunnel is removed and the original BIER packet is forwarded.

This work [MLM20a] has been published and presented at the International Conference on Software Defined Systems (SDS) in 2020 (see Appendix 2.2). In addition, the results of this work have been brought to standardization in the form of Internet drafts [MM19] [CML+22].

2.2.1.3 P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast

This section summarizes the research results from Merling et al. [MLM20b]. This publication is part of the additional content of this thesis. Therefore, it is only briefly summarized. Further details can be found in the associated publication and in the following section where the hardware implementation of BIER is described.

2.2.1.3.1 Research Objective The objective of this work was the implementation of BIER and BIER-FRR in P4 for the software switch BMv2 [p4l19]. At the time of this work, it was unclear whether BIER's iterative processing method could be implemented in P4. The first prototype was the result of my Master thesis [Lin19]. After its finalization, I continued with concept enhancement and the development of an improved prototype which was the basis for this publication.

2.2.1.3.2 Summary When a BFR receives a BIER packet, it iteratively serves all required next-hops indicated by the BIER bitstring. Conceptually, this could be done by iteratively serving the least significant activate bit. However, P4 does not support loops. Therefore, we leveraged the concept of recirculation. With recirculation, a packet is placed at the beginning of the P4 processing pipeline instead of being transmitted through an egress port. Therefore, a packet can be processed multiple times to implement a loop-based processing logic. When a BFR receives a BIER packet, it selects the next-hop for its least significant activated bit through a MAT. Then, the packet is cloned. The original packet is forwarded to the selected next-hop and the packet copy is recirculated so that it can be processed a second time. Further, BIER-FRR via unicast tunnels was implemented as explained in Section 2.2.1.2.

This work has been published as a journal paper in Journal of Network and Computer Applications (JNCA) [MLM20b] (see Appendix 2.3) in 2020. The source code of the BIER prototype has been published as open-source implementation on Github[4].

2.2.1.4 Hardware-Based Evaluation of Scalable and Resilient Multicast With BIER in P4

This section summarizes and presents the research results from Merling et al. [MLM21a]. This publication is part of the core content of this thesis. Further details can be found in the associated publication. First, the research objective is introduced. Then, the solution and evaluation results are explained.

2.2.1.4.1 Research Objective Although the prototype of [MLM20b] demonstrated that BIER can be implemented in P4, it was still unclear whether BIER can be efficiently supported in high-performance hardware. Software switches such as the BMv2 [p4l19]

[4]`https://github.com/uni-tue-kn/p4-bier`

have almost unlimited flexibility regarding the complexity of packet processing as everything is done entirely in software. Consequently, they only provide throughput in the order of a few Gb/s, which is unsuitable for network operators. In contrast, hardware targets like the Intel Tofino™ ASIC provide high throughput but have restrictions on the number of operations that can be applied per packet. Further, Intel Tofino™ implements the Tofino Native Architecture (TNA) whereas the software switch BMv2 implements a different architecture. This work aimed to implement and evaluate BIER and BIER-FRR for the Intel Tofino™ ASIC.

2.2.1.4.2 Implementation Concept Figure 2.5 illustrates the iterative BIER forwarding of a packet that has to be forwarded to three neighboring BFRs.

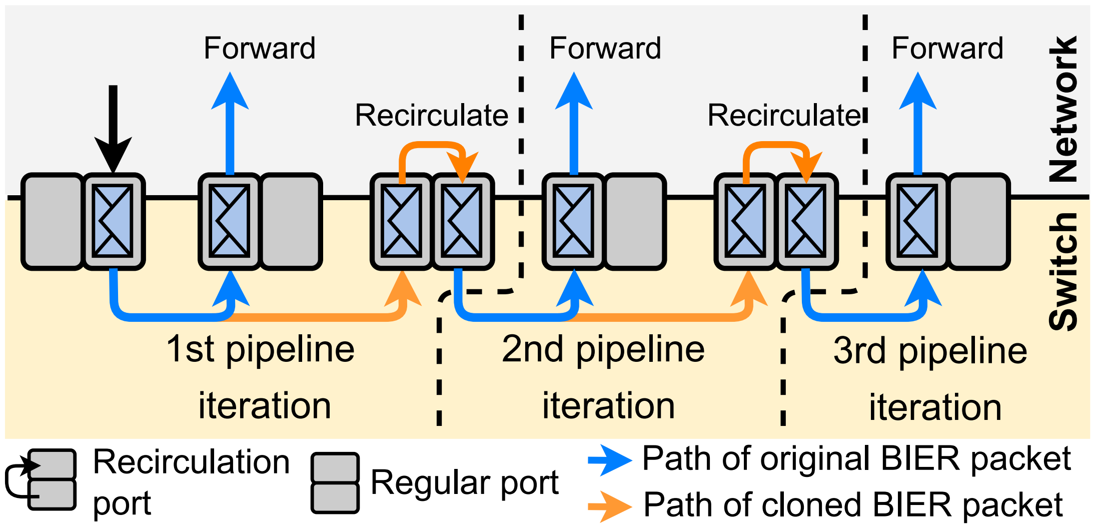


Figure 2.5: Iterative BIER processing. Figure from [MLM21a].

In each pipeline iteration, the next-hop for the least significant activate bit in the BIER bitstring is determined through a MAT. Afterwards, the packet is cloned to a so-called recirculation port[5] which allows processing in a second pipeline iteration. The already processed bits in this iteration are cleared from the BIER bitstring in the packet copy. The original packet is sent to the determined next-hop, and the packet copy is processed similarly until no activated bits are left in the BIER bitstring.

With this approach, a BIER packet with n next-hops requires $n-1$ packet copies and recirculations. However, each port of the Intel Tofino™ 1 can only process 100 Gb/s.

[5]A recirculation port is a port in loopback mode. When a packet reaches the end of the egress processing pipeline of a recirculation port, it is immediately placed in its ingress again for further processing.

Hence, when 100 Gb/s incoming BIER traffic should be replicated to three neighboring BFRs, 200 Gb/s additional traffic has to be processed on the recirculation port, which exceeds its processing capacity. To solve this problem, we turn physical ports into recirculation ports. This provides an additional 100 Gb/s processing capacity per port. However, these ports cannot be used for any other traffic. When performing the clone operation, we use a round-robin-based method to utilize all available recirculation ports evenly.

2.2.1.4.3 Evaluations We evaluate the throughput of the BIER P4 implementation on the Edgecore Wedge 100 BF-32X [Edg21] which is based on the Intel Tofino™ 1. Details on the hardware setup can be found in [MLM21a].

We send 100 Gb/s BIER traffic with a different number of next-hops and available recirculation ports to the Intel Tofino™ and measure the achieved throughput at the last receiver. The last receiver has the highest drop probability when insufficient processing capacity is available. Figure 2.6 illustrates the results.

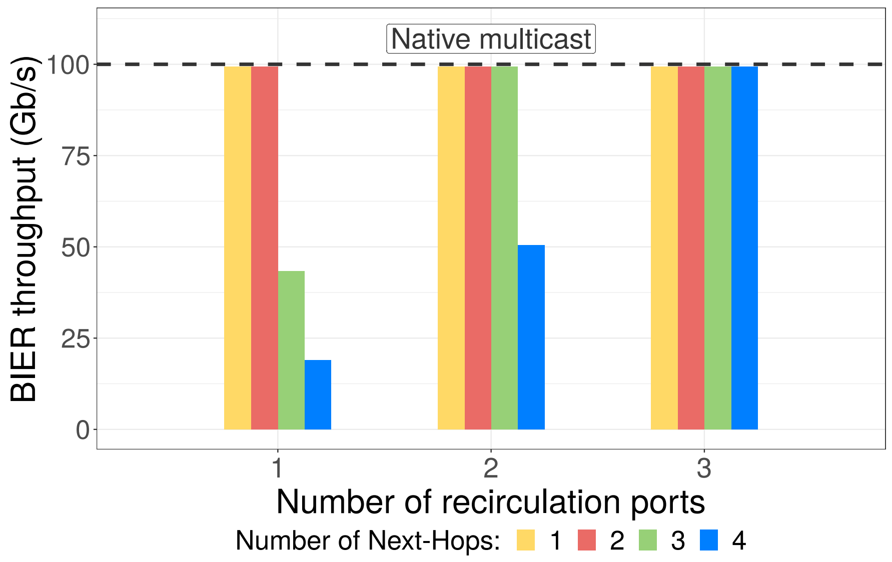


Figure 2.6: End-to-end BIER throughput on the last receiver with 100 Gb/s incoming BIER traffic. Figure from [MLM21a].

We compare the end-to-end throughput with native IPMC. With native multicast, line rate, i.e., 100 Gb/s, can be achieved independent of the number of recirculation ports and next-hops. With a single recirculation port up to $n = 2$ next-hops can be served with line rate, as a packet with n next-hops requires $n - 1$ recirculations. A single recirculation port does not provide sufficient processing capacity for $n > 2$ next-hops. Packets are dropped if the processing capacity is exceeded which leads to reduced end-to-end throughput with an increasing number of next-hops. With three recirculation ports, up to $n = 4$ next-hops can be served with line rate.

In reality, BIER traffic may take only a small fraction of the overall traffic. To that end, we perform simulations to determine the required number of additional[6] recirculation ports with a varying fraction $a = \{1, 2.5, 5, 10\}\%$ of BIER traffic and a varying number of next-hops for 100 Gb/s incoming traffic. Figure 2.7 illustrates the results.

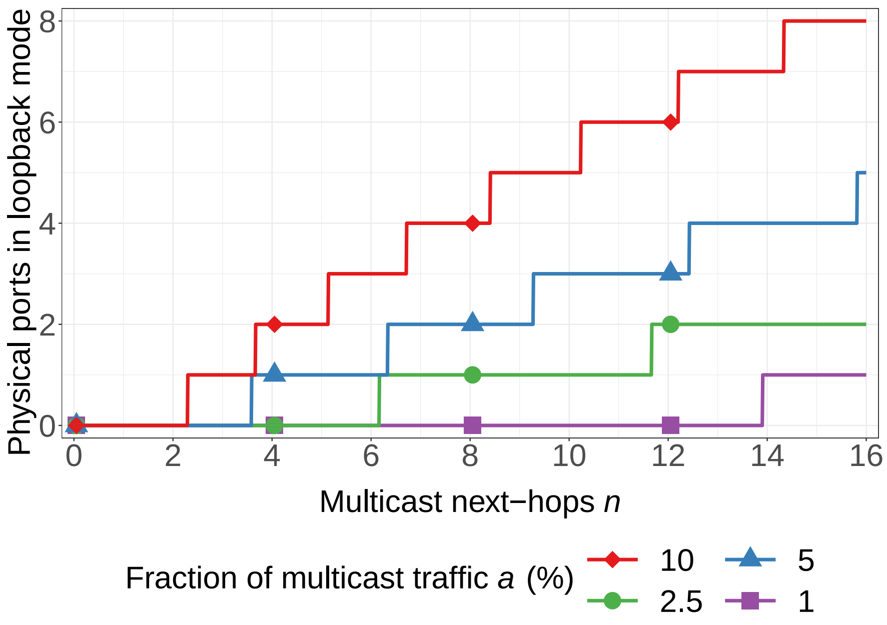


Figure 2.7: Number of required additional physical ports in loopback mode with a varying fraction of BIER traffic and a varying number of next-hops. Figure from [MLM21a].

For a small fraction of BIER traffic (1%), the internal recirculation port is sufficient to serve up to 13 next-hops. With an increasing number of next-hops and BIER traffic, more recirculation capacity and additional recirculation ports are required. Therefore, depending on the overall fraction of BIER traffic, only a few additional recirculation ports may be required to prevent packet loss for BIER-based multicast. Additional evaluations can be found in [MLM21a].

[6]Internal recirculation ports are not counted as additional ports.

2.2.1.4.4 Conclusion & Discussion We implemented BIER and BIER-FRR on the Intel Tofino™ and evaluated its end-to-end throughput. Hardware experiments showed that the implementation is able to support line rate processing. Further, we presented simulations to determine the required number of additional recirculation ports depending on the fraction of BIER traffic. If the fraction of BIER traffic is high, several physical ports have to be used as recirculation ports to guarantee line rate processing. These ports cannot be used for regular unicast traffic.

The results have been presented at IETF 108 in the BIER working group and have been published as a journal paper in IEEE Access [MLM21a] (see Appendix 1.2) in 2021.

2.2.1.5 Learning Multicast Patterns for Efficient BIER Forwarding with P4

This section summarizes and presents the research results from Lindner et al. [LMM23]. This publication is part of the core content of this thesis. Further details can be found in the associated publication. First, the research objective is introduced. Then, the solution and evaluation results are explained.

2.2.1.5.1 Research Objective With [MLM21a], we presented a P4-based BIER prototype that is able to forward BIER traffic at line rate. To do so, the implementation turned physical ports into recirculation ports to provide enough processing capacity to iteratively serve one next-hop after another. If not enough processing capacity is provided, BIER packets may be lost. The goal of this work was to reduce the required number of recirculations to save processing capacity and, therefore, to increase the efficiency of the P4-based BIER implementation on the Intel Tofino™ ASIC.

2.2.1.5.2 Concept Most P4 architectures, such as the TNA [Int21], define the concept of switch-intern static multicast groups. A switch-intern static multicast group contains a set of egress ports to which a packet is replicated without recirculation. Such a static multicast group is configured through the control plane. We leverage this concept to replicate a BIER packet to multiple next-hops without recirculation, which saves processing capacity. However, a BIER packet may need to be replicated to any subset of available next-hops / egress ports. To serve each subset of ports with a static multicast group, 2^{32} different static multicast groups are required on a 32 port switch, which exceeds the available number of static multicast groups[7]. Therefore, we propose to divide the ports of a switch in so-called configured port clusters $\mathcal{C} = \{C_1, ..., C_n\}$ such that all configured port clusters together cover all ports of the switch. All combinations of ports within a configured port cluster are installed as static multicast groups. On a 32 port switch with three configured port clusters of size $\{11, 11, 10\}$, this requires at most $2^{11} \ + 2^{11} \ + 2^{10} = 5.120$ static multicast groups, which is well feasible on the Intel Tofino™.

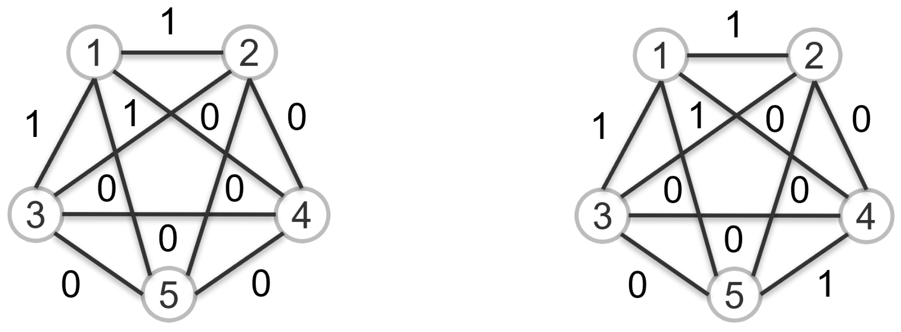
When a BIER packet is received, the set of configured port clusters that need to be served is determined based on the BIER bitstring. Then, for each selected configured port cluster, an appropriate static multicast group is used to replicate the packet to all required next-hops within this configured port cluster without recirculation. The selected configured port clusters are iteratively served through recirculation. Consequently, with k configured port clusters, a BIER packet with n next-hops is recirculated at most $k-1$ times instead of $n-1$ times, which significantly reduces the number of recirculations.

2.2.1.5.3 Optimization With k randomly selected configured port clusters[8], at most $k - 1$ recirculations are required per BIER packet. However, multicast traffic (and therefore BIER traffic) that utilizes random ports is unrealistic as end-users in certain regions may have common interests and share the same multicast groups, e.g., in case of IPTV. Consequently, some ports of a switch are more likely to be used together for a BIER packet than others, which we call port correlation. If a BIER packet requires only a few different configured port clusters, i.e., it is likely that all ports of a BIER packet are within a single or at most two configured port clusters, then the average number of recirculations is smaller than $k - 1$. Hence, the average number of recirculations is low when ports within a configured port cluster have a high port correlation and ports in different configured port clusters have a low port correlation.

We optimize the selection of configured port clusters through clustering, an unsupervised machine-learning method. To that end, we sample BIER packets at a switch and inspect their BIER bitstring. Based on the BIER bitstring, we derive the set of egress ports that need to be served for this BIER packet. We embed this information in a graph structure where each node represents a port of a switch. The graph is fully connected,

[7]Intel Tofino ™ 1 support at most 2^{16} static multicast groups [Int21].
[8]Such that all of them together cover all ports of a switch.

and the edges have weights. All weights are initially zero. Figure 2.8a and Figure 2.8b illustrate how two sampled BIER packets with ports {1, 2, 3} and {4, 5} are embedded on a 5-port switch.



(a) The edge weights between {1, 2}, {1, 3}, and {2, 3} are increased by one.

(b) The edge weight between {4, 5} is increased by one.

Figure 2.8: A full-mesh graph is augmented by port information from sampled packets: high edge weights indicate port pairs that frequently occur together in a BIER packet. Figures from [LMM23]. ©2022 IEEE.

For each port pair in a BIER packet, the corresponding edge in the graph embedding is increased by one. As a consequence, ports with high correlation, i.e., they are frequently used together to serve all next-hops of a BIER packet, have high edge weights. Ports with lower correlation have lower edge weights. We apply clustering methods based on Spectral Clustering [vL07] to build clusters, such that ports within clusters have high edge weights, and ports in different clusters have small edge weights. Cluster selection is further constrained by an upper bound on the available number of static multicast groups. Details on the clustering algorithms can be found in [LMM23].

2.2.1.5.4 Evaluations We evaluate the efficiency of three different proposed port clustering algorithms. The first algorithm is based on randomly selected configured port clusters (R). The second algorithm is a clustering method based on Spectral Clustering (PCSC). The last algorithm is an enhanced clustering algorithm based on Spectral Clustering (RPCO) that also allows to build overlapping clusters, i.e., some ports are part of multiple clusters. We generate artificial BIER traffic[9] with 4.5 next-hops on average and different port correlations $p \in \{0.7, 0.9, 0.99\}$. Large values of p indicate strong port correlation, and lower values of p indicate less port correlation. Further, we restrict the number of configurable static multicast groups m_{max}. The original BIER

[9]For details on the traffic model, see [LMM23].

forwarding algorithm of [MLM20b] and [MLM21a] corresponds to $m_{max} = 0$. The experiment is conducted 100 times, and average values are reported. Figure 2.9 illustrates the results.

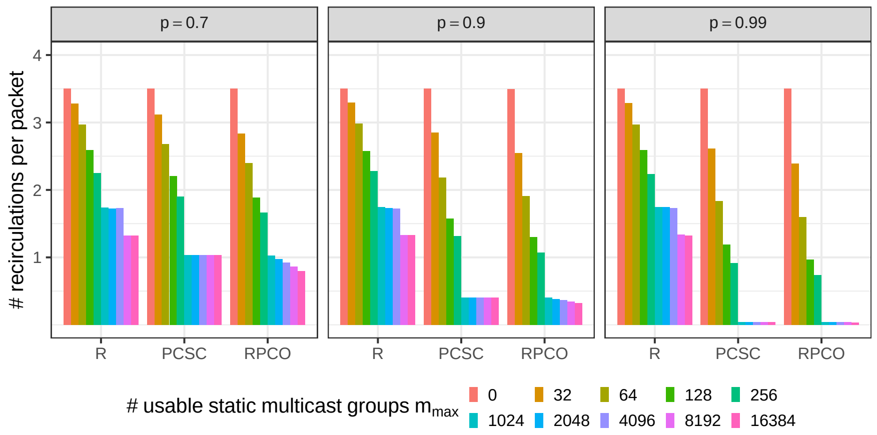


Figure 2.9: Average number of recirculations per packet for BIER traffic with 4.5 next-hops on average. Figure from [MLM21a]. ©2022 IEEE.

BIER traffic with 4.5 next-hops on average requires 3.5 recirculations per packet without static multicast groups ($m_{max} = 0$). The average number of recirculations per packet decreases with an increasing number of usable static multicast groups for all algorithms and port correlations p. The advanced algorithms (PCSC & RPCO) outperform the random clustering algorithm and are able to eliminate almost all recirculations with high port correlation and $m_{max} \geq 1024$. Advanced evaluations can be found in [LMM23].

2.2.1.5.5 Conclusion & Discussion We proposed an advanced BIER forwarding mechanism that leverages static multicast groups to reduce the required number of recirculations to save processing capacity. Further, unsupervised machine learning techniques have been used to optimize the selection of so-called configured port clusters. Simulations and hardware experiments show that the proposed mechanisms greatly reduce the required number of recirculations. The mechanism has been implemented on the Intel Tofino™ ASIC and runs at line rate.

The results have been presented at IETF 115 in the BIER working group and have been published as a journal paper in IEEE Transactions on Network and Service Management (TNSM) [LMM23] (see Appendix 1.4) in 2023. The source code of the Tofino-based prototype has been published as an open-source implementation on Github[10].

2.2.2 Data Transmission with QoS Requirements

This section summarizes the research results of this thesis on data transmission with QoS requirements. It comprises four works. Two works are part of the core content of this thesis, and two are part of the additional content. First, foundations of data transmission with QoS requirements are given. Then, the four works are summarized.

2.2.2.1 Foundations

Communication networks usually offer BE service, i.e., all data packets are treated the same. Therefore, data backups may experience the same end-to-end delay as Voice over IP (VoIP) telephony. If traffic is temporarily delayed due to network congestion, the backup is not affected, but VoIP telephony may no longer be possible in a meaningful way. These different demands are called Quality of Service (QoS) requirements. Traffic scheduling in combination with active queue management (AQM) tackles this problem by replacing the default first-in-first-out (FIFO) queue behavior with more advanced queuing and scheduling decisions, e.g., real-time traffic may be placed in another queue than common network traffic.

A recent approach to support data transmission with QoS requirements is TSN. TSN is an enhancement of Ethernet and provides various mechanisms to guarantee QoS requirements, e.g., traffic shaping and policing. The sender in a TSN domain is called talker whereas the receiver is called listener. Before a talker transmits its data, it signals its QoS requirements to the network. The network then configures itself in a distributed manner or through a central component such that the QoS requirements can be fulfilled.

2.2.2.2 Alternative Best Effort (ABE) for Service Differentiation: Trading Loss versus Delay

This section summarizes and presents the research results from Lindner et al. [LPM23]. This publication is part of the core content of this thesis. Further details can be found

[10]https://github.com/uni-tue-kn/rpco

in the associated publication. First, the research objective is introduced. Then, the solution and evaluation results are explained.

2.2.2.2.1 Research Objective Increasing buffer sizes in network devices lead to temporary long end-to-end delays [GN11], which may break the QoS requirements of some applications. The differentiated services framework (DiffServ) [BBC$^+$98] was introduced to allow for service differentiation in IP networks. With DiffServ, some traffic may be forwarded with higher priority to achieve its QoS requirements whereas lower priority traffic may receive worse service than without DiffServ. In this work, we revisited the idea of Alternative Best Effort (ABE) [HLTK01], which introduces a bounded-delay service class next to BE. A unique property of this service class is that low-delay forwarding of ABE traffic must not impede BE traffic. The goal of this work was to design and implement a novel packet scheduler in the Linux network stack that allows low-delay forwarding for ABE traffic without degrading the performance of BE traffic.

2.2.2.2.2 Concept We propose Deadlines, Saved Credits, and Decay (DSCD) as a novel packet scheduler for ABE and BE traffic. With DSCD, ABE traffic is forwarded with a significantly lower delay than BE but may experience higher packet loss. In contrast, BE traffic is forwarded as usual and receives the same service as in a pure BE system. A DSCD-based network device utilizes two FIFO queues, one to enqueue BE traffic and one to enqueue ABE traffic. Further, it leverages a so-called credit queue and two class-specific credit counters. Figure 2.10 illustrates the concept of DSCD.

Packets are assigned to a specific service class, i.e., ABE or BE, by the sender with the DiffServ code point (DSCP)[11]. When a packet is received by a DSCD-based network device, it is enqueued in an ABE or BE queue, depending on the DSCP value. Further, ABE packets are equipped with a configurable deadline T_d which gives an upper bound on the experienced queuing delay. When a packet is enqueued, a so-called credit element is created and placed in the credit queue ❶. A credit element preserves the right to send a number of bytes. To that end, it consists of a size and a service class, i.e., a 1500 byte ABE packet creates a 1500 byte credit element with service class ABE. A packet with size s is dequeued when its class-specific credit counter has at least value s ❷. When both credit counters are too low to dequeue a packet, a credit element is removed from the credit queue, and its associated credit counter is increased by its size

[11]DSCP is set through the ToS field in the IP header.

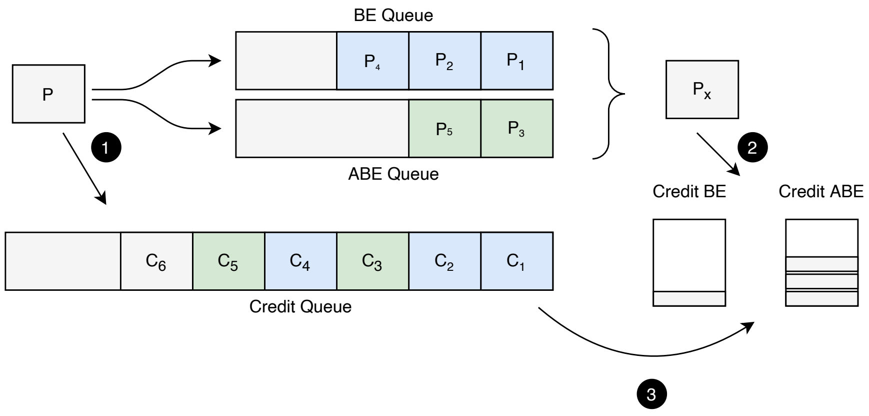


Figure 2.10: Concept of DSCD. An incoming packet is enqueued in its class-specific queue. If enough service credit is available, the packet is dequeued. ABE packets that exceed their maximal delay are dropped. Figure from [LPM23]. ©2022 IEEE.

➌. ABE packets that exceed their deadline are dropped, but their credit remains in the system for some time. Therefore, subsequent ABE packets can be served earlier than their BE counterparts. The FIFO order of the credit queue ensures that BE packets are no longer delayed than in a pure BE system. Hence, low-delay forwarding of ABE traffic does not impede BE traffic, as ABE packets utilize the transmission rights of previously dropped ABE packets.

2.2.2.2.3 Credit Devaluation Stored credit from dropped ABE packets may remain in the system for an infinite time. This may incentivize users to send unnecessary data to provoke packet loss and accumulate credit for later use. Further, credit should only be used during congestion periods and vanish afterwards. To that end, stored credit c is devaluated exponentially over time according to Equation 2.8, where Δt is the elapsed time since the last devaluation and λ is the devaluation rate.

$$c = c \cdot e^{-\lambda \cdot \Delta t} \tag{2.8}$$

The devaluation rate λ is configured through its half-life time $t_h = \frac{ln(2)}{\lambda}$, i.e., after one half-life time t_h only half the credit is still available.

2.2.2.2.4 Implementation in the Linux Network Stack We implemented DSCD in the Linux network stack as queuing discipline (QDisc)[12]. To that end, we developed an efficient approximation of the exponential function for the credit devaluation as the Linux kernel only supports[13] integer arithmetic. Further, a bandwidth estimation algorithm has been developed that works well even at moderate link utilization. Details on these implementations can be found in [LPM23].

We evaluate the efficiency of the DSCD implementation through the following experiment. We send TCP traffic in a 100 Gb/s environment and measure its achieved TCP goodput and the CPU load on the DSCD device. We compare the results of DSCD with various other existing QDiscs in the Linux kernel. Table 2.1 shows the results.

Table 2.1: TCP goodput and CPU load of various Linux QDiscs. Table from [LPM23]. ©2022 IEEE.

QDisc	TCP goodput (Gb/s)	CPU load (%)
DSCD	89.08	36.27
FQ-CoDel	89.02	38.99
FQ-PIE	89.00	44.21
SFQ	89.03	38.72
pfifo	89.06	35.41

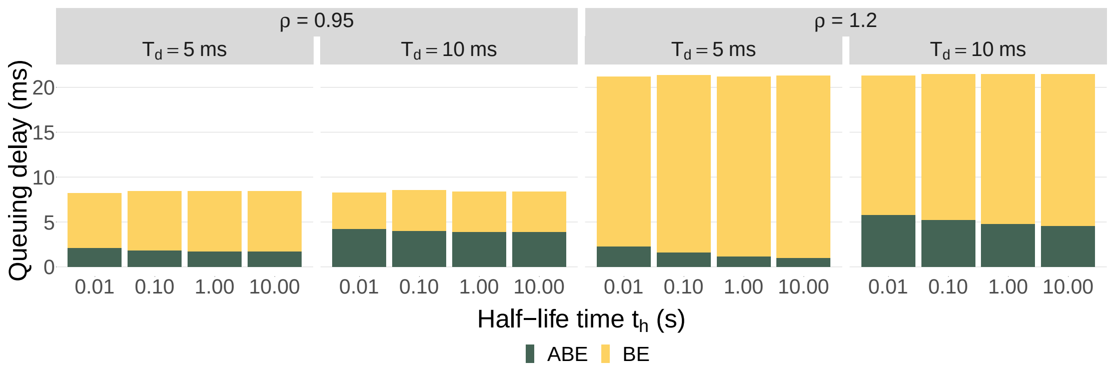
All considered QDiscs achieve around 89 Gb/s TCP goodput which is less than 100 Gb/s due to header overhead. DSCD and pfifo have the lowest CPU load with 36.27% and 35.41%. The results show that the DSCD implementation is efficient and comparable to other Linux QDiscs.

2.2.2.2.5 Evaluations We study DSCD's impact on packet loss and queuing delay for ABE and BE traffic. To that end, we send non-adaptive traffic, i.e., traffic that does not react to congestion signals, to investigate DSCD's properties without the influence of adaptive protocols such as TCP. We model the network traffic such that it includes traffic bursts. A traffic burst is a period of time where data is sent at higher rates than usual, which may lead to congestion in the network. We generate traffic with an offered load of $\rho \in \{0.95, 1.2\}$, where 90% of the traffic is labeled as BE and 10% as ABE.

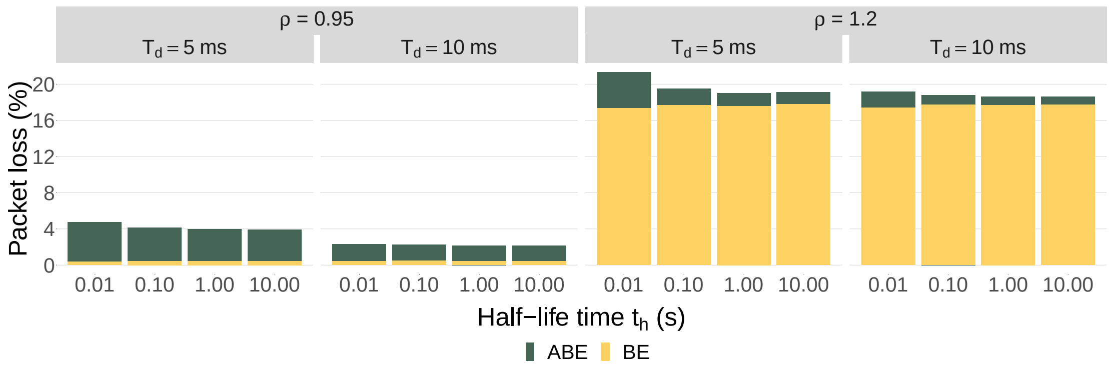
[12]QDiscs are implemented in the C programming language and perform tasks such as traffic shaping, packet classification, or packet dropping. They are located in the Linux kernel space and can perform almost arbitrary operations on packets.

[13]Floating point operations either require context switches to the user space or manually saving and restoring floating point registers.

With $\rho = 0.95$, traffic is sent on average with 95% of the bottleneck bandwidth but may exceed the bottleneck bandwidth during a burst period. With $\rho = 1.2$, the network is overloaded most of the time. Further, we vary the half-life time $t_h \in \{0.01, 0.1, 1, 10\}$ s and the maximal queuing delay $T_d \in \{5, 10\}$ ms. Details on the testbed setup can be found in [LPM23]. Figures 2.11a and Figure 2.11b illustrate the experienced queuing delay and packet loss for ABE and BE traffic.



(a) Queuing delay.



(b) Packet loss.

Figure 2.11: Queuing delay and packet loss for non-adaptive traffic with bursts. Figures from [LPM23]. ©2022 IEEE.

For $\rho = 0.95$, both ABE and BE experience only a small queuing delay (see Figure 2.11a). ABE experiences significantly less queuing delay than BE but faces higher packet loss as shown in Figure 2.11b. With $\rho = 1.2$, both ABE and BE face high packet loss as the network is constantly overloaded. BE traffic is queued for around 20 ms whereas ABE experiences at most 5 ms queuing delay. Higher values of T_d lead to higher queuing delay as packets remain longer in the queue. A higher half-life time t_h leads to less queuing delay and packet loss as stored credit is devaluated at a lower rate. The experiment validates DSCD‘s core property, i.e., that ABE is forwarded with low delay at the expense of higher packet loss whereas BE traffic receives the same service as in a pure BE system. Advanced evaluations that take adaptive transport protocols and different fractions of ABE and BE traffic into account can be found in [LPM23].

2.2.2.2.6 Conclusion & Discussion We proposed a novel packet scheduler called Deadlines, Saved Credits, and Decay (DSCD) that allows low-delay forwarding for ABE traffic without impeding BE traffic. It is based on the concept of credit elements to preserve the right to send a certain amount of bytes. DSCD was implemented in the Linux network stack and side products of the implementation were an efficient approximation of the exponential function and a bandwidth estimation algorithm that even works at moderate link utilization. Extensive hardware-based evaluations with up-to-date protocol stacks confirm DSCD‘s core properties.

This work has been published as a journal paper in IEEE Transactions on Networking (ToN) [LPM23] (see Appendix 1.3) in 2023. The source code of the DSCD implementation has been published as an open-source implementation on Github[14].

2.2.2.3 RAP Extensions for the Hybrid Configuration Model

This section summarizes the research results from Osswald et al. [OLWM21]. This publication is part of the additional content of this thesis. Therefore, it is only briefly summarized. Further details can be found in the associated publication.

2.2.2.3.1 Research Objective The Resource Allocation Protocol (RAP) is used in fully distributed TSN domains to signal stream reservation requests. In addition to the fully distributed configuration model, TSN supports the fully centralized configuration model, and the centralized network/distributed user, also called hybrid, configuration model. The fully centralized configuration model leverages other means to signal stream reservation requests. The goal of this work was to analyze and extend RAP such that it can be used in the hybrid configuration model.

2.2.2.3.2 Summary A talker in a TSN domain sends a so-called Talker Announce Attribute (TAA) to announce its stream requirements to the network. Similarly, the listener sends a so-called Listener Attach Attribute (LAA). Analysis of the included information in the TAA and LAA showed that RAP lacks some important properties which

[14]https://github.com/uni-tue-kn/dscd-linux-qdisc

are required to support all available TSN mechanisms in a TSN domain which uses the hybrid configuration model. For example, the earliest/latest transmit time cannot be signaled which is required for the Time-Aware Shaper (TAS) [80216] mechanism. Subsequently, it is proposed to extend RAP such that the missing information is included. Further, Osswald et al. [OLWM21] propose a software architecture for a Centralized User Configuration (CUC) component.

This work [OLWM21] has been published and presented at the International Conference on Emerging Technologies and Factory Automation (ETFA) in 2021 (see Appendix 2.4).

2.2.2.4 A Survey of Scheduling Algorithms for the Time-Aware Shaper in Time-Sensitive Networking (TSN)

This section summarizes the research results from Stüber et al. [SOLM23]. This work is part of the additional content of this thesis. Therefore, it is only briefly summarized. Further details can be found in the associated work.

2.2.2.4.1 Summary The Time-Aware Shaper (TAS) [80216] enables the transmission of real-time traffic with strict QoS requirements, e.g., zero queuing delay and jitter. In TSN, packets are enqueued in one of up to eight transmission queues depending on their priority. With TAS, so-called gates control which transmission queue is eligible for transmission at a given time, which is called schedule. The calculation of such a schedule is NP-complete [Ste10]. This work surveys the currently[15] available literature for TSN schedule computation. Research works are categorized and compared according to their modeling assumptions and open problems in the research field are identified. This work has been published as a journal paper in IEEE Access [SOLM23] (see Appendix Appendix 2.7) in 2023.

2.2.2.5 P4TG: 1 Tb/s Traffic Generation for Ethernet/IP Networks

This section summarizes the research results from Lindner et al. [LHM23]. This work is part of the core content of this thesis. Further details can be found in the associated publication. First, the research objective is introduced. Then, the solution and evaluation results are explained.

[15]Until March 2023.

2.2.2.5.1 Research Objective Traffic generators (TGs) are widely used to evaluate the performance of novel network equipment and protocols. To that end, they generate layer-2 (L2)/layer-3 (L3), i.e., Ethernet and IP, packets and measure the achievable transmission rates and other metrics. TGs are either software-based or hardware-based. Software-based TGs are highly flexible and customizable to individual needs but provide only support for low data rates. Hardware-based TGs support high data rates and generate traffic precisely, but are very expensive, up to several tens of thousands of dollars for 100 Gb/s support. Some offer additional functionality, e.g., means to debug and verify optical connectivity. In contrast, an Intel Tofino™ 1 with 3.2 Tb/s processing capacity costs less than eight thousand dollars[16].

This work aimed to design and implement a low-cost traffic generator based on the Intel Tofino™ that can generate up to 1 Tb/s traffic, including a wide range of measuring capabilities.

2.2.2.5.2 P4TG Overview P4TG is a 1 Tb/s hardware-based TG based on the Intel Tofino™ ASIC. P4TG provides two different modes: generation and analysis. In generation mode, P4TG can generate up to 10x 100 Gb/s network traffic. Traffic is sent through specified output ports and may be fed back to its input ports, possibly through other equipment. P4TG measures the transmission and reception rates and records packet loss, out-of-order packets, round trip time (RTT)[17], inter-arrival times (IATs), frame sizes, and frame types. Further, random traffic can be generated. In analysis mode, P4TG forwards traffic received on an input port to an output port and similarly analyses the received traffic. P4TG can be configured through a web-based interface.

2.2.2.5.3 Implementation Concept Intel Tofino™ comes with integrated traffic generation capabilities. Thereby, an interval can be specified in which a given byte sequence should be generated. We leverage this functionality and augment it with extensive measuring capabilities in the data plane. To that end, a generated packet is equipped with a special P4TG header, as shown in Figure 2.12.

The P4TG header consists of a sequence number used to identify out-of-order and lost packets, a transmission timestamp used to determine the RTT, and a stream identification field used to identify the corresponding stream.

[16]As of January 2023.

[17]The RTT is the time between the transmission of a packet on an output port and its reception on an input port.

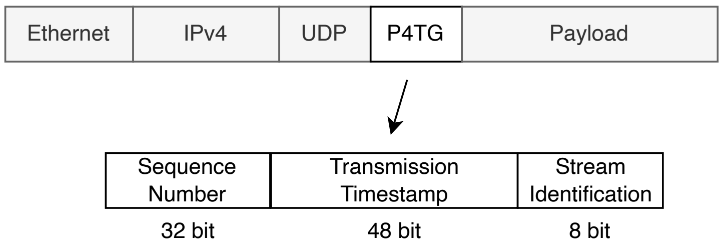


Figure 2.12: Generated packets are equipped with an Ethernet, IPv4, UDP, and P4TG header. Figure from [LHM23].

The transmitted and received bytes on layer-1 (L1)[18] and L2 are accumulated in a 64-bit register per port. They are used together with precise hardware timestamps to calculate transmission rates. Likewise, counters for detected out-of-order and lost packets are incremented in the data plane. Further, RTTs and IATs are sampled by the control plane to create statistics. Implementation details can be found in [LHM23].

2.2.2.5.4 Evaluation We evaluate P4TG's rate generation accuracy on a single port[19] and compare it with the software-based TG TRex [TRe22] and the hardware-based TG EXFO FTB-1 Pro [EXF19]. The prototype is implemented on the Edgecore Wedge 100 BF-32X [Edg21] which is based on the Intel Tofino™ 1. We configure a target rate R^{L1}_{target} on the TGs and measure the achieved relative L1 rate. Table 2.2 shows the results.

Table 2.2: Measured L1 rate relative to target rate R^{L1}_{target}. Table from [LHM23].

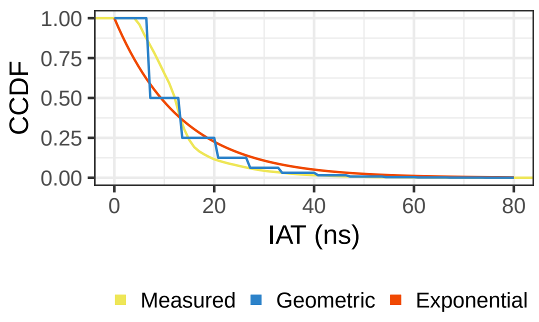
Gen. method	f^{L2}_{size} (byte)	R^{L1}_{target} (Gb/s)				
		0.001	10	50	75	100
P4TG	64	100.00	100.00	99.38	99.37	99.98
	256	100.00	99.80	99.92	99.77	99.92
TRex	64	99.99	100.00	100.00	91.30	68.48
	256	100.00	99.96	99.94	83.66	61.92
EXFO	64	100.00	100.00	100.00	100.00	100.00
	256	100.00	100.00	100.00	100.00	100.00

P4TG is capable of generating up to 100 Gb/s with a minimal frame size of $f^{L2}_{size} = 64$ byte with high accuracy. In contrast, the software-based TG TRex saturates[20] at around

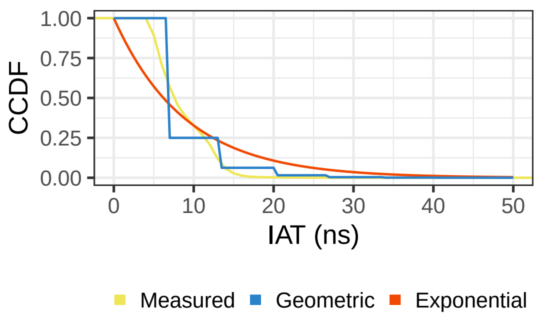
[18]Frame sizes on L1 include a preamble and inter-frame gap (additional 20 bytes).
[19]P4TG can generate traffic on up to 10 ports.
[20]The experiment was conducted on a VM with 10 Intel(R) Xeon(R) Gold 6134 CPU cores.

68 Gb/s for 64-byte frames. The hardware-based commercial TG EXFO precisely generates the configured target rates.

P4TG further supports the generation of random traffic, i.e., traffic with exponential distributed IATs. To that end, traffic is generated at line rate and dropped with a configured probability p, which resembles a geometric distribution[21]. We send traffic with target rates $R^{L1}_{target} \in \{50, 75\}$ Gb/s, sample the IATs, and compare the distribution of the samples with the theoretical geometric distribution and the corresponding exponential distribution. Figure 2.13a and 2.13b show the complementary cumulative distribution function (CCDF) of the sampled IATs.



(a) $R^{L1}_{target} = 50$ Gb/s



(b) $R^{L1}_{target} = 75$ Gb/s

Figure 2.13: CCDF of IATs with random traffic with 64 byte frames. Figures from [LHM23].

Both CCDFs match with the theoretically expected distributions. The deviation to the exponential distribution can be explained by the fact that IATs are bounded by the smallest possible frame size whereas they are unbounded in the case of the exponential distribution. Further evaluations can be found in [LHM23].

2.2.2.5.5 Conclusion & Discussion We presented P4TG, a 1 Tb/s traffic generator based on the Intel Tofino™. P4TG measures data rates, out-of-order packets, packet loss, RTTs, IATs, frame sizes, and frame types directly in the data plane for the highest possible accuracy. Further, it supports the generation of random traffic and allows external traffic analysis. P4TG comes with clearly lower costs than other hardware-based TGs. It may also be suited for traffic generation at even higher rates (up to 400 Gb/s per port) with Intel Tofino™ 2 and Intel Tofino™ 3.

[21]The geometric distribution is the discrete approximation of the exponential distribution.

This work has been published as a journal paper in IEEE Access [LHM23] (see Appendix 1.5) in 2023. The source code of the P4TG implementation has been published as open-source implementation on Github[22].

[22]https://github.com/uni-tue-kn/P4TG

3 Additional Scientific Work

This chapter summarizes additional scientific contributions, which have been made during my doctoral studies besides the publications presented in Chapter 2.

3.1 Workshop Organization

In 2020, I co-organized the 2nd KuVS Fachgespräch Network Softwarization and the 1st ITG Workshop on IT Security (ITSec) at the Eberhard Karls University Tübingen. Both workshops aimed to bring researchers and industry experts together to discuss the latest advances in network softwarization and IT security. The workshops comprised several keynote speeches, technical sessions, and panel discussions. Due to the COVID-19 pandemic, the workshops were held virtually. Based on the success of the 2nd KuVS Fachgespräch Network Softwarization in 2020, we also hosted the 3rd KuVS Fachgespräch Network Softwarization where I was again part of the organization team.

3.2 Research Proposals

I was involved in the creation of the DFG research proposal ”Resilient Communication with Programmable Hardware (ReCoPro)” for the priority program ”Resilience in Connected Worlds – Mastering Failures, Overload, Attacks, and the Unexpected (Resilient Worlds)” (SPP 2378) which has been granted at the end of 2022. Further, I was part of the management team for the Collaborative Project KITOS (support code 16KIS1161) at our chair and was responsible for technical reports and project deliverables.

3.3 Thesis Supervision

I supervised five Master theses and four Bachelor theses during my doctoral studies. Topics included the design and implementation of P4 programs and algorithms, literature overviews for topics currently outside the scope of the chair, and concepts and time plans for future teaching courses. The titles of the supervised theses are listed below.

B.Sc. Comparison and Analysis of Data Center Routing with RIFT and OSPF

M.Sc. Implementation and Evaluation of an Alternative Best Effort Traffic Scheduler in the Linux Network Stack

M.Sc. Design and Implementation of a RAP Prototype for the Centralized Network/Distributed User Model in TSN

B.Sc. Entwurf und Analyse eines Software-Projektes für das Tübinger Teamprojekt: Implementierung eines DNS Servers

B.Sc. Design and Implementation of a Web-Based Visualization Tool for Network Planning in Time-Sensitive Networking

M.Sc. Autonomous Integration of TSN-Unaware Applications with QoS Requirements in TSN Networks

M.Sc. Implementierung von Per-Stream Filtering and Policing für Time-Sensitive Networking auf einem 100G-fähigen Switch mithilfe von P4

B.Sc. Design und Implementierung einer generischen Control Plane GUI für P4-programmierbare Switche

Four of the supervised Master theses have laid the foundation for the start of independent Ph.D. topics of new co-workers.

3.4 Miscellaneous

During my doctoral studies, I supervised the lectures ”Grundlagen des Internets” (4 times) and ”Programmierprojekt” (4 times). I gave lectures on selected topics in our course ”Network Softwarization” (2 times). Further, I co-organized a weekly research seminar where younger co-workers presented their research progress and new research ideas were discussed.

In 2021, I co-authored the IETF Internet draft ”BIER Fast ReRoute” [CML$^+$22] of the BIER working group that may be adopted as an active working group document.

From 2021-2023, I was a reviewer for the following international journals, magazines, conferences, and workshops:

- Journal of Optical Communications and Networking (2021)
- International Conference on High-Performance Switching and Routing (HPSR 2021)
- International Conference on the Design of Reliable Communication Networks (DRCN 2021)
- IEEE Transactions on Network and Service Management (2020, 2022, 2023)
- IEEE/IFIP Network Operations and Management Symposium (NOMS 2020)

Finally, I gave several talks about current research topics at national workshops, academic salons, and the IETF. The titles of the talks are listed below.

- S. Lindner, M. Haeberle, D. Merling, and M. Menth: P4-Protect: 1+1 Path Protection for P4, 2nd KuVS Fachgespräch Network Softwarization, Tübingen, Deutschland, April 2020.
- H. Chen, M. McBride, S. Lindner, M. Menth, A. Wang, G. Mishra, Y. Liu, Y. Fan, L. Liu, X. Liu: BIER Fast Reroute, IETF 111, Juli 2021.
- D. Merling, S. Lindner, M. Menth: Robust LFA Protection for Software-Defined Networks (RoLPS), 3rd KuVS Fachgespräch ”Network Softwarization”, April 2022.
- S. Lindner, D. Merling, M. Menth: P4-Based Implementation of BIER and BIER-FRR for Efficient Multicast, Academic Salon on Low-Latency Communication, Programmable Network Components and In-Network Computation, September 2022.
- S. Lindner, D. Merling, and M. Menth: Efficient P4-based BIER Implementation on Tofino, IETF 115, November 2022.
- S. Lindner, G. Paradzik, M. Menth: Alternative Best Effort (ABE) for Service Differentiation: Trading Loss versus Delay, IETF 116, March 2023.

Bibliography

[80216] IEEE Standard for local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic. IEEE Standard, IEEE Computer Society, 2016.

[AZ08] A. Atlas and A. Zinin. RFC5286: Basic Specification for IP Fast Reroute: Loop-Free Alternates. Request for comments, Internet Engineering Task Force, September 2008. `http://www.rfc-editor.org/rfc/rfc5286.txt`.

[BBC+98] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. RFC2475: An Architecture for Differentiated Services. Internet-draft, Internet Engineering Task Force, December 1998. `https://www.rfc-editor.org/rfc/rfc2475`.

[BDG+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communications Review (CCR)*, 44(3):87–95, July 2014. `https://doi.org/10.1145/2656877.2656890`.

[CKR+21] Marco Chiesa, Andrzej Kamisiński, Jacek Rak, Gábor Rétvári, and Stefan Schmid. A Survey of Fast-Recovery Mechanisms in Packet-Switched Networks. *IEEE Communications Surveys & Tutorials*, 23(2):1253–1301, 2021. `https://doi.org/10.1109/COMST.2021.3063980`.

[CML+22] Huaimo Chen, Mike McBride, Steffen Lindner, Michael Menth, Aijun Wang, Gyan Mishra, Yisong Liu, Yanhe Fan, Lei Liu, and Xufeng Liu. BIER Fast ReRoute. Internet-draft, Internet Engineering Task Force, April 2022. Work in Progress. `https://datatracker.ietf.org/doc/draft-chen-bier-frr/05/`.

[Edg21] Edge-Core Networks. Wedge100BF-32X/65X Switch. `https://www.edge-core.com/_upload/images/Wedge_100-32X_DS_R04_20170615.pdf`, 2021. Accessed on 2023-01-07.

[EXF19] EXFO. FTB-1v2/FTB-1 Pro Platform. `https://www.exfo.com/umbraco/surface/file/download/?ni=10900&cn=en-US&pi=5404`, 2019. Accessed on 17.01.2023.

[GN11] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark Buffers in the Internet. *ACM Queue*, 9(11):40–54, November 2011. `https://doi.org/10.1145/2063166.2071893`.

[HHM+23] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with P4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications (JNCA)*, 212, 2023. `https://doi.org/10.1016/j.jnca.2022.103561`.

[HLTK01] P. Hurley, J.-Y. Le Boudec, P. Thiran, and M. Kara. ABE: Providing a Low-Delay Service within Best Effort. *IEEE Network Magazine*, 15(3):60–69, May 2001. `https://doi.org/10.1109/65.923942`.

[Int21] Intel. P416 intel tofino native architecture - public version. `https://github.com/barefootnetworks/Open-Tofino`, 2021. Accessed on 2023-01-08.

[ipe] iperf3. `http://software.es.net/iperf/`. Accessed on 27.01.2023.

[LHH+20] Steffen Lindner, Marco Häberle, Florian Heimgaertner, Naresh Nayak, Sebastian Schildt, Dennis Grewe, Hans Loehr, and Michael Menth. P4 In-Network Source Protection for Sensor Failover. In *International Workshop on Time-Sensitive and Deterministic Networking (TENSOR)*, pages 791–796, June 2020.

[LHM23] Steffen Lindner, Marco Häberle, and Michael Menth. P4TG: 1 Tb/s Traffic Generation for Ethernet/IP Networks. *IEEE Access*, 11:17525–17535, February 2023. ©2023 IEEE. Reprinted with permission. `https://doi.org/10.1109/ACCESS.2023.3246262`.

[Lin19] Steffen Lindner. Design and P4-Based Implementation of Scalable and Resilient BIER and BIER-TE for Large IP Multicast Networks. Master's thesis, University of Tuebingen, Sand 13, 72076 Tübingen, March 2019.

[LMHM20] Steffen Lindner, Daniel Merling, Marco Häberle, and Michael Menth. P4-Protect: 1+1 Path Protection for P4. In *P4 Workshop in Europe*, pages 21–27, December 2020. `https://doi.org/10.1145/3426744.3431327`.

[LMM23] Steffen Lindner, Daniel Merling, and Michael Menth. Learning Multicast Patterns for Efficient BIER Forwarding with P4. *IEEE Transactions on Network and Service Management*, 20(2):1238–1253, June 2023. ©2022 IEEE. Reprinted with permission. `https://doi.org/10.1109/TNSM.2022.3233126`.

[LPM23] Steffen Lindner, Gabriel Paradzik, and Michael Menth. Alternative Best Effort (ABE) for Service Differentiation: Trading Loss Versus Delay. *IEEE/ACM Transactions on Networking*, 31(4):1642–1656, August 2023. ©2022 IEEE. Reprinted with permission. `https://doi.org/10.1109/TNET.2022.3221553`.

[LPS$^+$13] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring Connectivity via Data Plane Mechanisms. In *USENIX Conference on Networked Systems Design & Implementation (NSDI)*, page 113–126, April 2013.

[MAB$^+$08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communications Review (CCR)*, 38:69–74, March 2008. `https://doi.org/10.1145/1355734.1355746`.

[MBM18] Daniel Merling, Wolfgang Braun, and Michael Menth. Efficient Data Plane Protection for SDN. In *IEEE Conference on Network Softwarization (NetSoft)*, pages 10–18, June 2018. `https://doi.org/10.1109/NETSOFT.2018.8459923`.

[MLM20a] Daniel Merling, Steffen Lindner, and Michael Menth. Comparison of Fast-Reroute Mechanisms for BIER-Based IP Multicast. In *International Conference on Software Defined Systems (SDS)*, pages 51–58, April 2020.

©2020 IEEE. Reprinted with permission. `https://doi.org/10.1109/SDS49854.2020.9143935`.

[MLM20b] Daniel Merling, Steffen Lindner, and Michael Menth. P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast. *Journal of Network and Computer Applications (JNCA)*, 169, November 2020. Reprinted with permission. `https://doi.org/10.1016/j.jnca.2020.102764`.

[MLM21a] Daniel Merling, Steffen Lindner, and Michael Menth. Hardware-Based Evaluation of Scalable and Resilient Multicast With BIER in P4. *IEEE Access*, 9:34500–34514, February 2021. ©2021 IEEE. Reprinted with permission. `https://doi.org/10.1109/ACCESS.2021.3061763`.

[MLM21b] Daniel Merling, Steffen Lindner, and Michael Menth. Robust LFA Protection for Software-Defined Networks (RoLPS). *IEEE Transactions on Network and Service Management (TNSM)*, 18(3):2570–2586, June 2021. ©2021 IEEE. Reprinted with permission. `https://doi.org/10.1109/TNSM.2021.3090843`.

[MM19] Daniel Merling and Michael Menth. BIER Fast Reroute. Internet-draft, Internet Engineering Task Force, March 2019. `https://datatracker.ietf.org/doc/draft-merling-bier-frr/00/`.

[MMWE18] Daniel Merling, Michael Menth, Nils Warnke, and Toerless. Eckert. An Overview of Bit Index Explicit Replication (BIER). In *IETF Journal*, March 2018.

[Moy98] John Moy. RFC2328: OSPF Version 2. Request for comments, Internet Engineering Task Force, April 1998. `https://www.rfc-editor.org/info/rfc2328`.

[NAC$^+$20] Nagendra Kumar Nainar, Rajiv Asati, Mach Chen, Xiaohu Xu, Andrew Dolganow, Tony Przygienda, Arkadiy Gulko, Dom Robinson, Vishal Arya, and Caitlin Bestler. BIER Use Cases. Internet-draft, Internet Engineering Task Force, September 2020. Work in Progress. `https://datatracker.ietf.org/doc/draft-ietf-bier-use-cases/12/`.

[OLWM21] Lukas Osswald, Steffen Lindner, Lukas Wüsteney, and Michael Menth. RAP Extensions for the Hybrid Configuration Model. In *IEEE International Conference on Emerging Technologies and Factory Automation*

(ETFA), pages 1–8, September 2021. ©2021 IEEE. Reprinted with permission. https://doi.org/10.1109/ETFA45728.2021.9613246.

[p4l19] p4lang. behavioral-model. https://github.com/p4lang/behavioral-model, 2019. Accessed on 2023-01-07.

[RHL06] Yakov Rekhter, Susan Hares, and Tony Li. RFC4271: A Border Gateway Protocol 4 (BGP-4). Request for comments, Internet Engineering Task Force, January 2006. https://www.rfc-editor.org/info/rfc4271.

[SOLM23] Thomas Stüber, Lukas Osswald, Steffen Lindner, and Michael Menth. A Survey of Scheduling Algorithms for the Time-Aware Shaper in Time-Sensitive Networking (TSN), June 2023. ©2023 IEEE. Reprinted with permission. https://doi.org/10.1109/ACCESS.2023.3286370.

[Ste10] Wilfried Steiner. An Evaluation of SMT-based Schedule Synthesis for Time-Triggered Multi-Hop Networks. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 375–384, November 2010. https://doi.org/10.1109/RTSS.2010.25.

[TRe22] TRex - Realistic Traffic Generator. https://trex-tgn.cisco.com, 2022. Accessed on 17.01.2023.

[vL07] U. v. Luxburg. A Tutorial on Spectral Clustering. *Statistics and Computing*, 17:395–416, August 2007.

[WRD$^+$17] IJsbrand Wijnands, Eric C. Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. RFC8279: Multicast Using Bit Index Explicit Replication (BIER). Request for comments, Internet Engineering Task Force, November 2017. https://www.rfc-editor.org/info/rfc8279.

Personal Contribution

Accepted Manuscripts (Core Content)

1. **P4-Protect: 1+1 Path Protection for P4** [LMHM20]

Scope of the joint work	This research work was done in the context of the research project „Future Internet Routing (FIR)“ funded by the Deutsche Forschungsgemeinschaft (DFG). The scope of this work was developing and implementing a scalable and efficient 1+1 protection mechanism.
Names of collaborators and their shares	Daniel Merling: Editorial assistance for writing the publication. Marco Häberle: Editorial assistance on publication and implementation support. Michael Menth: Scientific supervision and editorial assistance on the publication.
Importance of own contributions to the joint work	Main developer for the design of resilience mechanism. Responsible for the implementation. Main author of the publication.

2. **Hardware-Based Evaluation of Scalable and Resilient Multicast With BIER in P4** [MLM21a]

Scope of the joint work	This research work was done in the context of the research project „Future Internet Routing (FIR)“ funded by the Deutsche Forschungsgemeinschaft (DFG). The scope of this work was the implementation and evaluation of a BIER and BIER-FRR prototype on the Intel Tofino™.

Names of collaborators and their shares	Daniel Merling: Main author of the publication, taking on most of the write-up and rewriting during the revision phase. Michael Menth: Scientific supervision and editorial assistance on the publication.
Importance of own contributions to the joint work	Implementation and evaluation of P4-based BIER and BIER-FRR on the Intel Tofino™. Editorial assistance and co-author of the publication.

3. **Alternative Best Effort (ABE) for Service Differentiation: Trading Loss versus Delay** [LPM23]

Scope of the joint work	This research work was done in the context of the research project „Congestion Management für Paket-basierte Kommunikationsnetze (CoMa) “ funded by the Deutsche Forschungsgemeinschaft (DFG). The scope of this work was the design and implementation of a novel packet scheduler in the Linux network stack that allows low-delay forwarding for ABE traffic without degrading the performance of BE traffic.
Names of collaborators and their shares	Gabriel Paradzik: Responsible for the implementation. Editorial assistance and co-author of the publication. Michael Menth: Scientific supervision, concept development, and editorial assistance on the publication.
Importance of own contributions to the joint work	Main author of the publication. Responsible for concept development and most of the write-up and rewriting during the revision phase.

4. **Learning Multicast Patterns for Efficient BIER Forwarding with P4** [LMM23]

Scope of the joint work	This research work was done in the context of the research project „Resilient Communication with Programmable Hardware (ReCo-Pro) “ funded by the Deutsche Forschungsgemeinschaft (DFG). The scope of this work was the design and implementation of a novel BIER processing scheme to reduce the required number of recirculations to save processing capacity on the Intel Tofino™.
Names of collaborators and their shares	Daniel Merling: Assistance on concept development. Co-author and editorial assistance on publication. Michael Menth: Scientific supervision and editorial assistance on the publication.
Importance of own contributions to the joint work	Main author of the publication. Responsible for concept development, implementation, and most of the write-up and rewriting during the revision phase.

5. **P4TG: 1 Tb/s Traffic Generation for Ethernet/IP Networks** [LHM23]

Scope of the joint work	This research work was done in the context of the „bwNET2020+ “ research project funded by the Ministry of Science, Research and the Arts Baden-Württemberg (MWK). The scope of this work was the design and implementation of a low-cost traffic generator based on the Intel Tofino™.
Names of collaborators and their shares	Marco Häberle: Editorial assistance on publication. Michael Menth: Scientific supervision and editorial assistance on the publication.
Importance of own contributions to the joint work	Main author of the publication. Responsible for concept development, implementation, and most of the write-up and rewriting during the revision phase.

Accepted Manuscripts (Additional Content)

6. **P4 In-Network Source Protection for Sensor Failover** [LHH+20]

Scope of the joint work	This research work was done in the context of the research project „Future Internet Routing (FIR)“ funded by the Deutsche Forschungsgemeinschaft (DFG). The scope of this work was the design and implementation of an in-network sensor failover mechanism.
Names of collaborators and their shares	Marco Häberle: Feedback on the structure and write-up of the manuscript. Florian Heimgärtner: Feedback on the structure and write-up of the manuscript. Naresh Nayak: Feedback on the structure and write-up of the manuscript. Sebastian Schildt: Feedback on the structure and write-up of the manuscript. Dennis Grewe: Feedback on the structure and write-up of the manuscript. Hans Loehr: Feedback on the structure and write-up of the manuscript. Michael Menth: Scientific supervision and editorial assistance on the publication.
Importance of own contributions to the joint work	Main author of the publication. Responsible for concept development, implementation, and most of the write-up.

7. **Comparison of Fast-Reroute Mechanisms for BIER-Based IP Multicast** [MLM20a]

Scope of the joint work	This research work was done in the context of the research project „Future Internet Routing (FIR)“ funded by the Deutsche Forschungsgemeinschaft (DFG). The scope of this work was the development and comparison of two approaches to protect BIER traffic with FRR mechanisms.
Names of collaborators and their shares	Daniel Merling: Main author of the publication, taking on most of the write-up and rewriting during the revision phase. Michael Menth: Scientific supervision and editorial assistance on the publication.
Importance of own contributions to the joint work	Editorial assistance on the publication and assistance during the development phase.

8. **P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast** [MLM20b]

Scope of the joint work	This research work was done in the context of the research project „Future Internet Routing (FIR)“ funded by the Deutsche Forschungsgemeinschaft (DFG). The scope of this work was the development of a P4-based prototype of BIER and BIER-FRR for the software switch BMv2.
Names of collaborators and their shares	Daniel Merling: Responsible for the developed concepts. Main author of the publication, taking on most of the write-up and rewriting during the revision phase. Michael Menth: Scientific supervision and editorial assistance on the publication.
Importance of own contributions to the joint work	Implementation of the developed concepts. Editorial assistance and co-author of the publication

9. **RAP Extensions for the Hybrid Configuration Model** [OLWM21]

Scope of the joint work	This research work was done in the context of the research project „KITOS " funded by the German Federal Ministry of Education and Research (BMBF). The scope of this work was the analysis and extension of RAP such that it can be used in the hybrid configuration model.
Names of collaborators and their shares	Lukas Osswald: Responsible for the developed concepts. Main author of the publication, taking on most of the write-up and rewriting during the revision phase. Lukas Wüsteney: Editorial assistance and co-author of the publication. Michael Menth: Scientific supervision and editorial assistance on the publication.
Importance of own contributions to the joint work	Editorial assistance and co-author of the publication.

10. **Robust LFA Protection for Software-Defined Networks (RoLPS)** [MLM21b]

Scope of the joint work	This research work was done in the context of the research project „Future Internet Routing (FIR)" funded by the Deutsche Forschungsgemeinschaft (DFG). The scope of this work was the development and implementation of a 1:1 protection mechanism based on LFAs.

Names of collaborators and their shares	Irene Müller-Benz: Implementation of an early prototype in her master thesis. Daniel Merling: Supervision of the master thesis of Irene Müller-Benz. Main author of the publication, taking on most of the write-up and rewriting during the revision phase. Michael Menth: Scientific supervision and editorial assistance on the publication.
Importance of own contributions to the joint work	Implementation of the developed concepts. Editorial assistance and co-author of the publication.

11. **A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research** [HHM+23]

Scope of the joint work	Literature study on the state-of-the-art of data plane programming with P4 covering fundamentals, advances, and applied research.
Names of collaborators and their shares	Frederik Hauser: Coordination of all activities around this writing project. Main author of the publication. Daniel Merling, Marco Häberle: Analysis, classification, and summarization of applied research work. Writing input and feedback on the foundation chapters. Help in structure definition and review of the complete manuscript. Vladimir Gurevich: Writing input and feedback on the foundation chapters. Florian Zeiger, Reinhard Frank: Feedback on the structure and write-up of the manuscript. Michael Menth: Scientific supervision of the research project. Editorial assistance on the publication.

Importance of own contributions to the joint work	Analysis, classification, and summarization of applied research works. Writing input and feedback on the foundation chapters. Help in structure definition and review of the complete manuscript.

12. **A Survey of Scheduling Algorithms for the Time-Aware Shaper in Time-Sensitive Networking (TSN)** [SOLM23]

Scope of the joint work	This research work was done in the context of the research project „KITOS " funded by the German Federal Ministry of Education and Research (BMBF). Literature study on the state-of-the-art of schedule computation for the Time-Aware Shaper (TAS) in Time-Sensitive Networking (TSN).
Names of collaborators and their shares	Thomas Stüber: Coordination of all activities around this writing project. Main author of the publication. Lukas Osswald: Editorial assistance and co-author of the publication. Michael Menth: Scientific supervision and editorial assistance on the publication.
Importance of own contributions to the joint work	Editorial assistance, feedback on the manuscript during write-up.

Publications

1 Accepted Manuscripts (Core Content)

1.1 P4-Protect: 1+1 Path Protection for P4

P4-Protect: 1+1 Path Protection for P4

Steffen Lindner
University of Tübingen
steffen.lindner@uni-tuebingen.de

Daniel Merling
University of Tübingen
daniel.merling@uni-tuebingen.de

Marco Häberle
University of Tübingen
marco.haeberle@uni-tuebingen.de

Michael Menth
University of Tübingen
menth@uni-tuebingen.de

ABSTRACT

1+1 protection is a method to secure traffic between two nodes against failures in between. The sending node duplicates the traffic and forwards it over two disjoint paths. The receiving node assures that only a single copy of the traffic is further forwarded to its destination. In contrast to other protection schemes, this method prevents almost any packet loss in case of failures. 1+1 protection is usually applied on the optical layer, on Ethernet, or on MPLS.

In this work we propose the application of 1+1 for P4-based IP networks. We define an 1+1 protection header for that purpose. We describe the behavior of sending and receiving nodes and provide a P4-based implementation for the Behavioral Model version 2 (bmv2) software switch and the hardware switch Tofino Edgecore Wedge 100BF-32X. We illustrate how to secure traffic, e.g. individual TCP flows, on the Internet with this approach. Finally, we present performance results showing that the P4-based implementation efficiently works on the Tofino Edgecore Wedge 100BF-32X.

KEYWORDS

p4, software defined networking, 1+1 protection

ACM Reference Format:
Steffen Lindner, Daniel Merling, Marco Häberle, and Michael Menth. 2020. P4-Protect: 1+1 Path Protection for P4. In *3rd P4 Workshop in Europe (EuroP4'20), December 1, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3426744.3431327

1 INTRODUCTION

There are various concepts to secure traffic transmission against failure of path components such as links or nodes. The fastest is 1+1 protection. A sender duplicates traffic and forwards it over disjoint paths while the receiver forwards only the first copy received for every packet. In case of a failure, any packet loss can be avoided, which makes 1+1 protection attractive for highly reliable applications. 1+1 protection is implemented in optical networks to protect an entire trunk. It is also available for MPLS [10] and Ethernet [9], which are carrier technologies for IP and introduce signaling complexity. In this paper, we leverage the P4 programming language [3] to provide 1+1 protection for IP networks. We program P4 switches such that they feature IP forwarding, the sending and receiving node behaviour of 1+1 protection which includes IP encapsulation and decapsulation. We call this approach P4-Protect. Targets of our implementation are the software switch BMv2 and the hardware switch Tofino Edgecore Wedge 100BF-32X. A particular challenge is the selection of the fist copy of every duplicated packet at the receiver. We provide a controller that allows to set up 1+1 protection between P4 nodes implementing P4-Protect. Furthermore, protected flows can be added using a fine-granular description based on various header fields. We evaluate the performance of P4-Protect on the hardware switch. We show that P4-Protect can be used with only marginal throughput degradation and we illustrate that P4-Protect can significantly reduce jitter when both paths have similar delays.

The paper is structured as follows. Section 2 gives an overview of related work. Section 3 describes the 1+1 protection mechanism used for our implementation and extensions for its use on the general Internet. Section 5 presents a P4-based implementation including specifics for the Tofino Edgecore Wedge 100BF-32X. We evaluate the performance of P4-Protect on the hardware switch in Section 6 and conclude the paper in Section 7.

2 RELATED WORK

We review various resilience concepts for communication networks. Afterwards, we give examples for 1+1 protection.

2.1 Overview

Rerouting reorganizes the traffic forwarding to avoid failed components. This happens on a time scale of a second. Fast reroute (FRR) locally detects that a next hop is unreachable and deviates traffic to an alternative next hop [1]. The detection may take a few 10s of milliseconds so that traffic loss cannot be avoided. Both rerouting and FRR do not utilize backup resources under failure-free conditions, but their reaction time suffers from failure detection delay. 1:1 protection leverages a primary/backup path concept. To switch over, the head-end node of the paths needs to be informed about a failure, which imposes additional delay. With restoration, recovery paths may be dynamically allocated so that even more time is needed to establish the restoration paths [19, p. 31]. 1+1 protection duplicates traffic and sends it over two disjoint paths whereby the receiving node needs to eliminate duplicates. That method is fastest, but it requires extra capacities also under failure-free conditions. Some services can afford short network downtimes, other services greatly benefit from 1+1 protection's high reliability.

The surveys [15], [20], and [7] provide an overview of various protection and restoration schemes. The authors of [7] discuss survivability techniques for non-WDM networks like automatic protection switching (APS) and self healing rings (SHR) as well as dynamic restoration schemes in SONET. They further describe protection methods for optical WDM networks. A comprehensive overview of protection and restoration mechanisms for optical, SONET/SDH, IP, and MPLS networks can be found in [19].

SDN with inband signalling increases the need for fast and local protection against failures because the controller may no longer be reachable in case of a failure or highly loaded. In addition, with SDN new protection mechanisms can be implemented, e.g., to reduce state in the network. Examples are given in [14].

2.2 1+1 Protection

At first we will look at standards with respect to 1+1 protection, followed by other work related to 1+1 protection.

2.2.1 Standards. The ITU-T specification Y.1703 [10] defines a 1+1 path protection scheme for MPLS. It adds sequence numbers to packets and replicates them on disjoint paths. At the end of the paths, duplicate packets are identified by the sequence number and eliminated. P4-Protect works similarly. However, it does not require MPLS. It is compatible with IP and works over the Internet.

802.1CB [8] defines a redundant transmission mode for Time-Sensitive Networking (TSN), called *Frame Replication and Elimination for Reliability* (FRER). Each packet of a stream is equipped with a sequence number, replicated, and then sent through two disjoint paths to a destination. Both destination and/or traversing nodes eliminate duplicate packets. FRER supports two algorithms: *VectorRecoveryAlgorithm* and *MatchRecoveryAlgorithm*. With the *VectorRecoveryAlgorithm*, an acceptance window is used to accept packets with higher sequence numbers than expected. With the *MatchRecoveryAlgorithm* all sequence numbers except the last seen are accepted, which is used to prevent misbehaviour. In-order delivery is currently out of scope in 802.1CB.

DetNet [5] provides capabilities to carry data flows for real-time applications with extremely low data loss rates and bounded latency within a network. *Packet Replication and Elimination* (PRE) is a service protection method for DetNet, which leverages the 1+1 protection concept. PRE adds sequence numbers or time stamps to packets in order to identify duplicates. Packets are replicated and sent along multiple different paths, e.g., over explicit routes. Duplicates are eliminated, mostly at the edge of the DetNet domain. The *Packet Ordering Function* can be used at the elimination point to provide in-order delivery. However, this requires extra buffering.

2.2.2 Other Work on 1+1 Protection. The authors of [21] compare several implementation strategies of 1+1 protection, i.e, traditional 1+1 path protection, network redundancy 1+1 path protection (diversity coding) [2], and network-coded 1+1 path protection. Their analytical results show that diversity coding and network coding can be more cost-efficient, i.e., they require about 5-20% less reserved bandwidth. The delay impact of 1+1 path protection in MPLS networks has been investigated in [17]. McGettrick et. al [13] consider 10 Gb/s symmetric LR-PON. They reveal switch-over times to a backup OLT of less than 4 ms. Multicast traffic has often real-time requirements. Mohandespour et. al extend the idea of unicast 1+1 protection to protect multicast connections [16]. They formulate the problem of minimum cost multicast 1+1 protection as a 2-connectivity problem and propose heuristics. Braun et. al [4] propose maximally redundant trees for 1+1 protection in BIER, a stateless multicast transport mechanism. It leverages the concept of multicast-only FRR [11].

3 P4-PROTECT: CONCEPT

We first give an overview of P4-Protect. We present its protection header, the protection connection context, and the operation of the Protection Tunnel Ingress (PTI) and Protection Tunnel Egress (PTE).

3.1 Overview

With P4-Protect, a protection connection is established between two P4 switches. Protected traffic is duplicated by a PTI node and simultaneously carried through two protection tunnels to a PTE node. The PTE receives the duplicated traffic and forwards the first copy received for every packet.

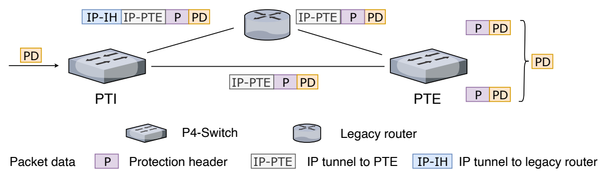


Figure 1: With P4-Protect, a PTI encapsulates and duplicates packets, and sends them over disjoint paths; the PTE decapsulates the packets and forwards only the first packet copy.

Figure 1 illustrates the protocol stack used with P4-Protect. The PTI adds to each packet received for a protected flow a protection header (P) that contains a sequence number which is incremented for each protected packet. The packet is equipped with an additional IP header (IP-PTE) with the PTE's IP address as destination. The PTI duplicates that packet and forwards the two copies over different paths. The paths may be different due to traffic engineering (TE) capabilities of the network or path diversity may be achieved through an additional intermediate hop. When the PTE receives a packet, it removes its outer IP header (IP-PTE). If the sequence number in the protection header is larger than the last sequence number received for this connection, it removes the protection header and forwards the packet; otherwise, the packet is dropped. The latter is needed as duplicate packets are also considered harmful.

3.2 Protection Header

The protection header contains a 24 bit Connection Identifier (CID), a 32 bit Sequence Number (SN) field, and an 8 bit *next protocol* field. The CID is used to uniquely identify a protection connection at the PTE. The sequence number is used at the PTE to identify duplicates. The *next protocol* field facilitates the parsing of the next header. We reuse the IP protocol numbers for this purpose.

3.3 Protection Connection Context

A protection connection is set up between a PTI and PTE. Their IP addresses are associated with this connection, including two interfaces over which duplicate packets are forwarded. For each connection, the PTI has a sequence number counter SN_{last}^{PTI} which is incremented for each packet forwarded over the respective protection connection. Likewise, the PTE has a variable SN_{last}^{PTE} which records the highest sequence number received for the respective protection connection. A CID is used to identify a connection at the PTE. A PTI may have several protection connections with the same CID but different PTEs (see Section 5.5.1).

3.4 PTI Operation

The PTI has a set of flow descriptors that are mapped to protection connections. If the PTI receives a packet which is matched by a specific flow descriptor, the PTI processes the packet using the corresponding protection connection. That is, it increments the SN_{last}^{PTI}, adds a protection header with CID, *next protocol* set to IPv4, and the SN set to SN_{last}^{PTI}. Then, an IP header is added using the PTI's IP address as source and the IP address of the PTE associated with the protection connection as destination. The packet is duplicated and forwarded over the two paths associated with the protection connection.

3.5 PTE Operation

During failure-free operation, the PTE receives duplicate packets via two protection tunnels. When the PTE receives a packet, it decapsulates the outer IP header. It uses the CID in the protection header to identify the protection connection and the corresponding SN_{last}^{PTE}. If the SN in the protection header is larger than SN_{last}^{PTE}, SN_{last}^{PTE} is updated by SN, the protection header is decapsulated, and the original packet is forwarded; otherwise, the packet is dropped.

The presented behavior works for unlimited sequence numbers. The limited size of the sequence number space makes the acceptance decision for a packet more complex. Then, a SN larger than SN_{last}^{PTE} may indicate a copy of a new packet, but it may also result from a very old packet. To solve this problem, we adopt the use of an acceptance window as proposed in [10]. The window is W sequence numbers large. Let SN_{max} be the maximum sequence number. If $SN_{last}^{PTE}+W < SN_{max}$ holds, a new sequence number SN is accepted if the following inequality holds:

$$SN_{last}^{PTE} < SN \leq SN_{last}^{PTE} + W \tag{1}$$

If $SN_{last}^{PTE} + W \geq SN_{max}$ holds, a new sequence number SN is accepted if one of the two following inequalities holds:

$$SN_{last}^{PTE} < SN \tag{2}$$

$$SN < SN_{last}^{PTE} + W - SN_{max} \tag{3}$$

This allows a packet copy to arrive $SN_{max} - W$ sequence numbers later than the corresponding first packet copy without being recognized as new packets.

AXE [12] tries to solve a similar problem, namely the de-duplication of packets. The hash of incoming packets is used to access special registers and associated header fields are stored. When another packet with the same hash arrives and the stored header fields match the incoming packet, the packet is a duplicate. No hash collision is considered. This technique detects duplicates quite reliably. However, AXE considers L2 flooding for learning bridges and therefore operates on relatively low bandwidths. P4-Protect must be able to de-duplicate several 100G connections, for the Tofino Edgecore Wedge 100BF-32X 3.2 Tb/s. Hence the AXE approach is not feasible due to the required register memory space and, depending on the hash algorithm, the high probability for hash collisions.

4 DISCUSSION

In this section the protection properties of P4-Protect are examined in more detail. Both, advantages and limitations of P4-Protect are discussed. The impact on jitter, packet loss and packet reordering are considered. To that end, we provide examples of traffic streams received by the PTE and their results after duplicate elimination.

4.1 Impact on Jitter

P4-Protect replicates packets to two preferable disjoint paths. If both paths suffer from jitter, P4-Protect can compensate the overall end-to-end jitter. Figure 2 illustrates the impact of P4-Protect on the overall end-to-end jitter. Packet 3 on path 1 has a very high delay due to jitter. As P4-Protect always forwards the first version of in-order packets, packet 3 is forwarded from the second path and thereby compensates the jitter delay.

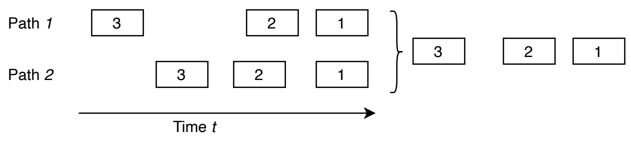


Figure 2: P4-Protect can reduce jitter.

4.2 Impact of Packet Loss

P4-Protect forwards the first version of a packet. If a path fails, all packet replicas of the other path are forwarded correctly. If individual packets are lost on one path, their replicas from the other path are not necessarily forwarded. This phenomenon ist illustrated in Figure 3. Four packets are replicated by the PTI and sent over two disjoint paths. The second path has a higher latency. As a result, packet 4 of the first path arrives before packet 3 on the second path. Now, SNE_{last}^{PTE} is set to 4, and as a consequence, packet 3 on the second path is discarded.

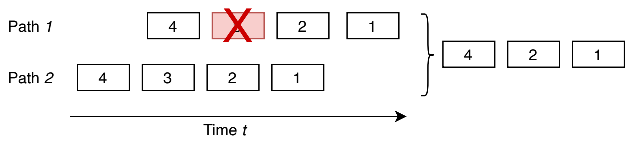


Figure 3: Packet loss may not be compensated by P4-Protect.

This behavior is due to the scalable design of P4-Protect. Only the last accepted sequence number is stored and checked at a new packet arrival. Missing packets are not memorized nor are packets buffered. This example clarifies that the objective of P4-Protect is to protect quickly against path failures, it is not to compensate for individual packet losses.

4.3 Packet Reordering

Packet reordering on a path has different sources, e.g., parallelism in network devices, link bundling, and special QoS configurations [18]. In case of packet reordering, P4-Protect may cause packet loss.

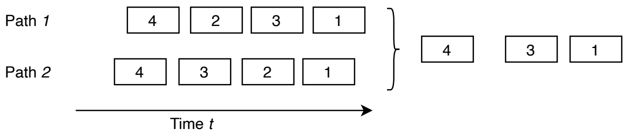


Figure 4: As it is not possible to check for lost packets, reordering leads to packet loss.

Figure 4 illustrates the impact of packet reordering. Path 1 has a slightly lower end-to-end latency than path 2. Due to packet reordering, the PTE receives the packets of path 1 in the order 1 3 2 4 instead of 1 2 3 4 . Moreover, packet 3 of the first path arrives slightly before packet 2 of the second path. As a result, the PTE accepts packets 1, 3 and 4 from path 1 and discards packet 2 from path 2. The main reason for this behavior is that P4-Protect does not memorize missing packets. Therefore, they cannot be accepted if they arrive in the wrong order. Limited arithmetic operations and storage access on our specific hardware target inhibit more sophisticated checks.

5 IMPLEMENTATION

In this section we present the implementation of P4-Protect. We describe the supported header stacks, explain the control blocks, their organization in ingress and egress control flow, and we reveal implementation details about some control blocks. Finally, we sketch most relevant aspects of the P4-Protect controller.

5.1 Supported Header Stacks

Incoming packets are parsed so that their header values can be accessed within the P4 pipeline. To that end, we define the following supported header stacks. Unprotected IP traffic has the structure IP/TP, i.e., IP header and some transport header (TCP/UDP), and protected IP traffic has the structure IP/P/IP/TP, i.e., the IP header with the PTE's address, the protection header, the original IP header, and a transport header. IP traffic without transport header is parsed only up to the IP header.

5.2 Control Blocks

We present three control blocks of our implementation of P4-Protect. They consider the packet processing by PTI and PTE.

5.2.1 *Control Block: Protect&Forward.*

When the PTI receives an IP packet, it is parsed and matched against the Match+action (MAT) table ProtectedFlows. In case of a match, the packet is equipped with an appropriate header stack, duplicated, and sent to appropriate egress ports. In case of a miss, the packet is processed by a standard IPv4 forwarding procedure.

5.2.2 *Control Block: Decaps-IP.*

When the PTE receives an IP packet with the PTE's own IP address, the IP header is decapsulated. If the next protocol indicates a protection header, the packet is handed over to the Decaps-P control block; otherwise, the packet is processed by the Protect&Forward control block since the resulting packet may need to be protected and forwarded.

5.2.3 *Control Block: Decaps-P.*

In the Decaps-P control block, the PTE examines the protection header and decides whether to keep or drop the packet as it is a copy of an earlier received packet. To keep the packet, the protection header is decapsulated.

5.3 Ingress and Egress Control Flow

The inter-dependencies between the control blocks suggest the following ingress control flow: Decaps-IP, Decaps-P, Protect&Forward. At a mere PTI, no action is performed by the Decaps-IP and Decap-P control block. The Protect&Forward takes care that protected traffic is duplicated and sent over two different paths and that unprotected traffic is forwarded by normal IPv4 operation. At a mere PTE, protected traffic is decapsulated and selected before being forwarded by normal IPv4 operation. Unprotected traffic is just forwarded by normal IPv4 operation.

5.4 Control Block Implementations

In the following, we explain implementation details of the Protect&Forward control block and the Decaps-P control block. We omit the Decaps-IP control block as it is rather simple.

5.4.1 *Protect&Forward Control Block.*

The operation of the Protect&Forward control block is illustrated in Figure 5. It utilizes the MAT ProtectedFlows to process all packets. It effects that protected traffic is encapsulated at the PTI with a protection header and an IP header for tunneling.

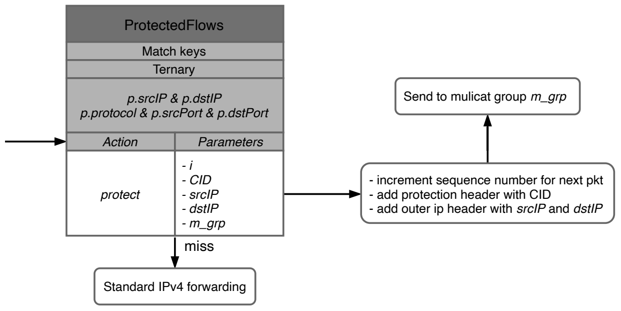

ProtectedFlows
Match keys
Ternary
p.srcIP & p.dstIP
p.protocol & p.srcPort & p.dstPort
Action
Parameters
protect
- i
- CID
- srcIP
- dstIP
- m_grp
miss
Standard IPv4 forwarding
- increment sequence number for next pkt
- add protection header with CID
- add outer ip header with srcIP and dstIP
Send to mulicat group m_grp

Figure 5: The MAT ProtectedFows inside the Protect&Forward control block is applied to IPv4 traffic.

The MAT ProtectedFlows uses a ternary match on the classic 5-tuple description of a flow: the source and destination IP address and port as well as the protocol field. In case of a match, the MAT maps a packet to a specifc protection connection and calls the protect action with the connection-specific parameters i, CID, $srcIP$, $dstIP$, and m_grp. The protect action increments the register $SN_{last}^{PTI}[i]$ where i is a connection-specific index to access a register containing the last sequence number. On the Tofino target, this is performed by a separate register action. The protect action further pushes a protection header on the packet including CID, i.e., the

CID, $SN_{last}^{PTI}[i]$, and the next protocol set to IPv4. Then, it pushes an IPv4 header with the IP address *srcIP* of the PTI as source IP and the IP address *dstIP* of the PTE as destination IP. The protocol field of this outer IP header is set to `P4-Protect`. Finally, the multicast group of the packet is set to *m_grp*. It is a connection-specific multicast group. It effects that the packet is duplicated and sent to two egress ports in order to deliver it via two protection tunnels to the PTE. In case of a miss, the packet is unprotected and handled by a standard IPv4 forwarding procedure, which is not further explained in this paper.

5.4.2 Decaps-P Control Block. The Decaps-P control block decides whether a packet is new and should be forwarded or dropped. It compares the sequence number SN of the packet's protection header with the last sequence number of the corresponding protection connection. The latter can be accessed by the register $SN_{last}^{PTE}[CID]$ where CID is given in the protection header. The acceptance is decided based on Equation (1) or Equation (3) depending on the value of SN and W where W is given as a constant.

As the check is rather complex, it requires careful implementation for the Tofino target [1]. It leverages the fact that we set $W = \frac{SN_{max}}{2}$. Furthermore, it requires a reformulation of Equation (1) and Equation (3).

If $W \leq SN$ holds, the following two inequalities must be met:

$$SN_{last}^{PTE} < SN \quad (4)$$

$$SN - SN_{last}^{PTE} \leq W \quad (5)$$

Otherwise, if $SN < W$, it is sufficient that only one of the following two inequalities holds:

$$SN_{last}^{PTE} < SN \quad (6)$$

$$W \leq SN_{last}^{PTE} - SN \quad (7)$$

Both cases are implemented as separate register actions on the Tofino target. With 32 bit sequence numbers, a minimum packet size of 40 bytes and a transmission speed of $C = 1$ Tb/s, a delay difference up to 1.6s can be compensated.

The bmv2 version of the implementation can be found at Github[2]. The Tofino version of the implementation can be found at Github[3] as well.

5.5 Controller for P4-Protect

P4-Protect's controller offers an interface for the management of protection connections and protected flows. It configures in particular the MAT ProtectedFlows but also other MATs needed for standard IPv4 forwarding or IP decapsulation. In the following, we explain the configuration of protection connections and protected flows.

5.5.1 Configuration of Protection Connections. A protection connection is established by choosing registers on PTI and PTE to record the last sequence numbers SN_{last}^{PTI} and SN_{last}^{PTE} of a protection connection. The connection identifier is the PTE's index to access SN_{last}^{PTE}. On the PTI, a different index i may be chosen to access SN_{last}^{PTI}. Furthermore, the registers are initialized with zero. Moreover, the controller sets up a multicast group *m_grp* for each connection so that its traffic will be replicated in an efficient way to the two desired interfaces.

5.5.2 Configuration of Protected Flows. A protected flow is established by adding a new flow rule in the MAT ProtectedFlows of the PTI. It contains an appropriate flow descriptor and the parameters to call the action protect. Those are the index i associated with the corresponding protection connection, the CID needed at the PTE to identify the protection connection, the IP address of the PTI, the IP of the PTE, and the multicast group *m_grp*.

6 EVALUATION

In this section we evaluate the performance of the implemented mechanism on the Tofino Edgecore Wedge 100BF-32X. First, we compare packet processing times with and without P4-Protect. Then, we demonstrate that very high data rates can be achieved with and without P4-Protect on a 100 Gb/s interface. Finally, we show that P4-Protect can provide a transmission service with reduced jitter compared to the jitter of both protection tunnels.

6.1 Packet Processing Time

P4-Protect induces forwarding complexity. To evaluate its impact, we leverage P4 metadata to calculate the time a packet takes from the beginning of the ingress pipeline to the beginning of the egress pipeline. This is sufficient for a comparison as all work for P4-Protect is done in the ingress pipeline and all considered forwarding schemes utilizes the same egress pipeline. We compare three forwarding modes: a plain IP forwarding implementation (plain), P4-Protect for unprotected traffic (unprotected), and P4-Protect for protected traffic (protected).

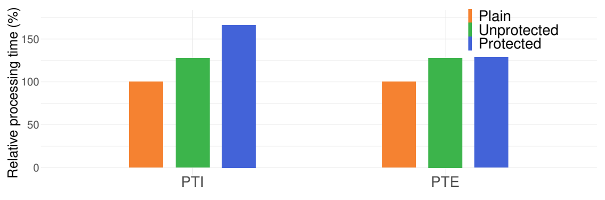


Figure 6: Ingress-to-egress packet processing time at PTI and PTE for three forwarding modes: plain, unprotected, and protected.

Figure 6 shows the ingress-to-egress packet processing time on both PTI and PTE for the three mentioned forwarding modes. The duration is given relative to the processing time for plain forwarding mode. We observe the lowest processing time at PTI and PTE for plain forwarding as it has the least complex pipeline. With P4-Protect, the processing time at both PTI and PTE is larger than with plain forwarding as the operations are more complex. At PTI, the processing time is even larger with protected forwarding (166%) than with unprotected forwarding (127%). At PTE, the processing times for protected and unprotected traffic are equal and 27% longer than with plain forwarding.

[1]Tofino is a high-performance chip which operates at 100 Gb/s so that only a limited set of operations can be performed for each packet, in particular in connection with register access.
[2]Repository: https://github.com/uni-tue-kn/p4-protect
[3]Repository: https://github.com/uni-tue-kn/p4-protect-tofino

In our implementations, we have used only a minimal IPv4 stack for all three forwarding modes. With a more comprehensive IPv4 stack, the relative overhead through P4-Protect is likely to be smaller.

6.2 TCP Goodput

We set up iperf3 connections between client/server pairs and measure their goodput. Each iperf3 connection consists of 15 parallel TCP flows. Two switches are bidirectionally connected via two 100 Gb/s interfaces. Four client/server pairs are connected to the switches via 100 Gb/s interfaces. Up to 4 clients download traffic from their servers via the trunk between the switches.

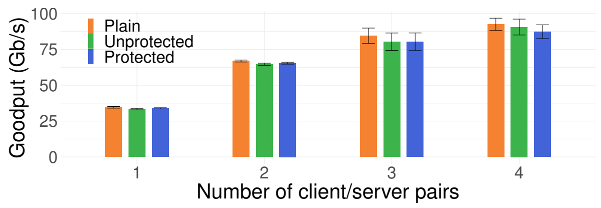


Figure 7: Impact of varying number of client/server pairs exchanging traffic with iperf3.

Figure 7 shows the overall goodput for a various number of client/server pairs, each transmitting traffic over a single TCP connection. The goodput is given for the forwarding modes plain, unprotected, and protected. We performed 20 runs per experiment and provide the 95% confidence interval.

A single, two, and, three TCP connections cannot generate sufficient traffic to fill the 100 Gb/s bottleneck link. However, with four TCP connections a goodput of around 90 Gb/s is achieved. This is less than 100% because of overhead due to Ethernet, IP, and TCP headers and due to the inability of TCP to efficiently utilize available capacity at high data rates. Most important is the observation that all three forwarding modes lead to almost identical goodput. The goodput for protected and unprotected forwarding is slightly lower than plain forwarding, which is apparently due to the operational overhead of P4-Protect.

6.3 Impact on Jitter

We examine the effect of 1+1 path protection on jitter. Two hosts are connected to two Tofino Edgecore Wedges 100BF-32X. The switches are connected with each other via two paths with intermediate Linux servers. Their interfaces are bridged and cause an artificial, adjustable, uniformly distributed jitter. We leverage the *tc* tool for this purpose [6]. All lines have a capacity of 100 Gb/s.

In our experiment, we send pings between the two hosts with and without P4-Protect. Figure 8 reports the average round trip time (RTT) deviation for the pings. Unprotected traffic suffers from all the jitter induced on a single path. Protected traffic suffers only from about half the jitter. This is because P4-Protect forwards the earliest received packet copy and minimizes packet delay occurred on both links.

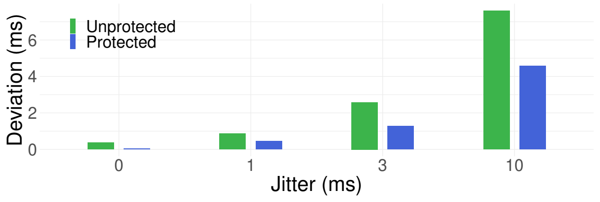


Figure 8: Impact of 1+1 protection on jitter.

7 CONCLUSION

In this paper we proposed P4-Protect for 1+1 path protection with P4. It may be utilized to protect traffic via two largely disjoint paths. We presented an implementation for the software switch bmv2 and the hardware switch Tofino Edgecore Wedge 100Bf-32X. The evaluation of P4-Protect on the hardware switch revealed that P4-Protect increases packet processing times only little, that high throughput can be achieved with P4-Protect, and that jitter is reduced by P4-Protect when traffic is carried over two path with similar delay but large jitter.

ACKNOWLEDGMENTS

The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/1-2. The authors alone are responsible for the content of the paper.

REFERENCES

[1] A. Atlas et al. 2008. RFC5286: Basic Specification for IP Fast Reroute: Loop-Free Alternates .
[2] Ender Ayanoglu et al. 1993. Diversity coding for transparent self-healing and fault-tolerant communication networks. *IEEE ToC* 41(11) (1993).
[3] P. Bosshart et al. 2014. P4: Programming Protocol-Independent Packet Processors. *ACM CCR* 44(3) (2014).
[4] Wolfgang Braun et al. 2017. Performance Comparison of Resilience Mechanisms for Stateless Multicast Using BIER. In *IFIP/IEEE*.
[5] Norman Finn, Pascal Thubert, Balazs Varga, and János Farkas. 2019. Deterministic Networking Architecture. RFC 8655. https://doi.org/10.17487/RFC8655
[6] Linux Foundation. 2019. *Linux Traffic Control*.
[7] A. Fumagalli et al. 2000. IP restoration vs. WDM protection: is there an optimal choice? *IEEE Network Magazine* 14(6) (2000).
[8] IEEE Computer Society. 2017. *Frame Replication and Elimination for Reliability*. Technical Report.
[9] ITU. 2006. ITU-T Recommendation G.803/Y.1342 (2006), Ethernet Protection Switching .
[10] ITU. 2010. ITU-T Recommendation G.7712/Y.1703 (2010), Internet protocol aspects – Operation, administration and maintenance.
[11] A. Karan et al. 2015. RFC7431: Multicast-Only Fast Reroute.
[12] James McCauley, Mingjie Zhao, Ethan J. Jackson, Barath Raghavan, Sylvia Ratnasamy, and Scott Shenker. 2016. The Deforestation of L2. In *Proceedings of the 2016 ACM SIGCOMM Conference*.
[13] Sèamas McGettrick et al. 2013. Ultra-fast 1+1 protection in 10 Gb/s symmetric Long Reach PON. In *IEEE ECOC)*.
[14] Daniel Merling et al. 2018. Efficient Data Plane Protection for SDN. IEEE (NetSoft).
[15] Christopher Metz. 2000. IP protection and restoration. *IEEE Internet Computing* 4(2) (2000).
[16] Mirzad Mohandespour et al. 2015. Multicast 1+1 protection: The case for simple network coding. In *IEEE ICNC*.
[17] Grazziela Niculescu et al. 2010. The Packet Delay in a MPLS Network Using "1+1 Protection. In *IEEE Advanced International Conference on Telecommunications*.
[18] Michal Przybylski, Bartosz Belter, and Artur Binczewski. 2005. Shall we worry about Packet Reordering? *Computational Methods in Science and Technology* 11.
[19] Jean Philippe Vasseur et al. 2004. *Network Recovery*. Morgan Kaufmann.
[20] Dongyun Zhou et al. 2000. Survivability in Optical Networks. *IEEE Network Magazine* 14(6) (2000).
[21] Harald Øverby et. al. 2012. Cost comparison of 1+1 path protection schemes: A case for coding. In *IEEE ICC*.

1.2 Hardware-Based Evaluation of Scalable and Resilient Multicast With BIER in P4

Received February 4, 2021, accepted February 19, 2021, date of publication February 24, 2021, date of current version March 4, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3061763

Hardware-Based Evaluation of Scalable and Resilient Multicast With BIER in P4

DANIEL MERLING, STEFFEN LINDNER, AND MICHAEL MENTH, (Senior Member, IEEE)
Chair of Communication Networks, University of Tuebingen, 72076 Tübingen, Germany

Corresponding author: Daniel Merling (daniel.merling@uni-tuebingen.de)

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under Grant ME2727/1-2. The authors alone are responsible for the content of this paper.

ABSTRACT Traditional IP multicast (IPMC) maintains state per IPMC group in core devices to distribute one-to-many traffic along tree-like structures through the network. This limits its scalability because whenever subscribers of IPMC groups change, forwarding state in the core network needs to be updated. Bit Index Explicit Replication (BIER) has been proposed by the IETF for efficient transport of IPMC traffic without the need of IPMC-group-dependent state in core devices. However, legacy devices do not offer the required features to implement BIER. P4 is a programming language which follows the software-defined networking (SDN) paradigm. It provides a programmable data plane by programming the packet processing pipeline of P4 devices. The contribution of this article is threefold. First, we provide a hardware-based prototype of BIER and BIER fast reroute (BIER-FRR) which leverages packet recirculation. Our target is the P4-programmable high-performance switching ASIC Tofino; the source code is publicly available. Second, we perform an experimental evaluation, with regard to failover time and throughput, which shows that up to 100 Gb/s throughput can be obtained and that failures affect BIER forwarding for less than 1 ms. However, throughput can decrease if switch-internal packet loss occurs due to missing recirculation capacity. As a remedy, we add more recirculation capacity by turning physical ports into loopback mode. To quantify the problem, we derive a prediction model for reduced throughput whose results are in good accordance with measured values. Third, we provide a provisioning rule for recirculation ports, that is applicable to general P4 programs, to avoid switch-internal packet loss due to packet recirculation. In a case study we show that BIER requires only a few such ports under realistic mixes of unicast and multicast traffic.

INDEX TERMS Software-defined networking, P4, bit index explicit replication, multicast, resilience, scalability.

I. INTRODUCTION

IP multicast (IPMC) has been proposed to efficiently distribute one-to-many traffic, e.g. for IPTV, multicast VPN, commercial stock exchange, video services, public surveillance data distribution, emergency services, telemetry, or content-delivery networks, by forwarding only one packet per link. IPMC traffic is organized in IPMC groups which are subscribed by hosts. Figure 1 shows the concept of IPMC. IPMC traffic is forwarded on IPMC-group-specific distribution trees from the source to all subscribed hosts. To that end, core routers maintain forwarding state for each IPMC group to determine the next-hops (NHs) of an IPMC packet. Scalability issues are threefold. First, a significant amount of storage is required to keep extensive forwarding state. Second, when subscribers of an IPMC group change, the distribution tree needs to be updated by signaling the changes to core devices. Third, the distribution trees have to

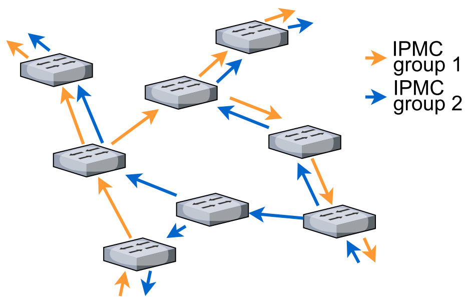


Figure 1. Two multicast distribution trees.

The associate editor coordinating the review of this manuscript and approving it for publication was Martin Reisslein.

be updated when the topology changes or a failure is detected. Therefore, traditional IPMC comes with significant management and state overhead. As a result, traditional IPMC is often avoided and multicast is implemented on the application layer. Thereby, one-to-many traffic is carried via network layer unicast, which is not efficient.

The IETF proposed Bit Index Explicit Replication (BIER) [1] for efficient transport of IPMC traffic. BIER introduces a BIER domain where core routers do not need to maintain IPMC-group-dependent state. Upon entering the BIER domain, IPMC packets are equipped with a BIER header which specifies all destinations of the packet within the BIER domain. The BIER packets are forwarded through the BIER domain towards their destinations on paths from the Interior Gateway Protocol (IGP), which we call 'routing underlay' in the following. Thereby, only one packet is forwarded per link. When the BIER packets leave the BIER domain, the BIER header is removed.

Unicast and BIER traffic may be affected by failures. IP-Unicast traffic is often protected by fast reroute (FRR) mechanisms for IP (IP-FRR). IP-FRR leverages precomputed backup entries to quickly reroute a packet on a backup path when the primary NH is unreachable. Tunnel-based BIER-FRR [2] is used to protect BIER traffic by tunneling BIER packets through the routing underlay. The tunnel may be also affected by a failure, but FRR or timely updates of the forwarding information base (FIB) in the routing underlay quickly restore connectivity. However, BIER is not supported by legacy devices and there is no dedicated BIER hardware available. P4 [3] is a programming language that follows the software-defined networking (SDN) paradigm for programming protocol-independent packet processors. P4 allows developers to write high-level programs to define the packet processing pipeline of programmable network devices. A target-specific compiler translates the P4 program for execution on a particular device. With the P4-programmable data plane new protocols can be implemented and deployed in short time.

In previous work [2], [4] we implemented BIER and tunnel-based BIER-FRR for the P4 software switch bmv2 [5]. However, the developers of the bmv2 clarify that the 'BMv2 is not meant to be a production-grade software switch' [5] and is, therefore, only a 'tool for developing, testing and debugging P4 data planes' [5]. Thus, it remains unclear whether BIER and BIER-FRR forwarding is simple enough to be implemented also on P4-capable hardware platforms which entail functional and runtime constraints to achieve high-speed forwarding.

The contribution of this article is threefold. First, we provide a new prototype for BIER and BIER-FRR on the P4-programmable switching ASIC Tofino [6] which is used in the Edgecore Wedge 100BF-32X [7], a 32 100 Gb/s port high-performance P4 switch, and make our code publicly available.

Second, we conduct an experimental performance study with regard to failover time and throughput. The evaluations show that connectivity can be restored within less than 1 ms and that a throughput of up to 100 Gb/s can be obtained. However, we observe reduced throughput under certain conditions and conjecture that this results from switch-internal packet loss due to missing recirculation capacity. We add more recirculation capacity by turning physical ports into loopback mode to avoid switch-internal packet loss in case of recirculation. To quantify the problem, we derive a prediction model for BIER throughput whose results are in good accordance with measured values.

Third, we propose a provisioning rule for recirculation ports to avoid switch-internal packet loss due to packet recirculation. It is applicable to general P4 programs and helps to avoid throughput reduction on outgoing links. Finally, we utilize the provisioning model to show in a case study that only a few ports in loopback mode suffice to avoid internal packet loss with BIER under realistic mixes of unicast and multicast traffic.

The paper is structured as follows. In Section II we describe related work. Section III contains a primer on BIER and tunnel-based BIER-FRR. Afterwards, we give an overview on P4 in Section IV and explain important properties. In Section V, we briefly describe the P4 implementation of BIER and tunnel-based BIER-FRR for the Tofino. Section VI contains our evaluation and the model for throughput prediction of BIER. In Section VII we present a model to provision recirculation ports. We conclude the paper in Section VIII.

II. RELATED WORK

First, we describe related work for SDN-based multicast in general. Then, we review work for BIER-based multicast. Finally, we present P4 projects that are based on packet recirculation.

A. SDN-BASED MULTICAST

Elmo [8] increases scalability of traditional IPMC in data center environments by leveraging characteristics of data center networks, in particular symmetric topologies and short paths. By encoding multicast group information in the packet header, this information is no longer stored in forwarding devices. This significantly reduces the dynamic state that needs to be maintained by core nodes.

Two surveys [9], [10] provide a comprehensive overview of SDN-based multicast. They review the development of traditional multicast and different aspects of SDN-based multicast, e.g., building of distribution trees, group management, and approaches to improve the efficiency of multicast. Most of the papers in the surveys discuss multicast mechanisms that are based on explicit IPMC-group-dependent state in core devices. The downsides of those traditional IPMC approaches have been discussed in Section I. We still discuss some papers on IPMC due to their efforts to make traditional IPMC more efficient. The papers often focus on intelligent tree building mechanisms that reduce the state, or efficient signaling techniques when IPMC groups or the topology changes. The surveys also consider works that utilize SDN to improve multicast. They are related as our approach also takes an SDN approach. Therefore, we present some representative examples from that area.

1) OPTIMIZATION OF MULTICAST TREES

Rückert *et al.* propose Software-Defined Multicast (SDM) [11]. SDM is an OpenFlow-based platform that provides well-managed multicast for over-the-top and overlay-based live streaming services tailored for P2P-based video stream delivery. The authors extend SDM in [12] with traffic engineering capabilities. In [13] the authors propose address translation from the multicast address to the unicast address of receivers at the last multicast hop in OpenFlow switches. This reduces the number of IPMC-group-dependent forwarding entries in some nodes.

Steiner trees are often used to build multicast distribution trees [14]. Several papers modify the original Steiner-tree problem to build distribution trees with minimal cost [15], number of edges [16], number of branch nodes [17], delay [18], or for optimal position of the multicast source [19].

The authors of [20] implement a multicast platform in OpenFlow with a reduced number of forwarding entries. It is based on multiple shared trees between different IPMC groups. The Avalanche Routing Algorithm (AvRA) [21] considers properties of the topology of data center networks to build trees with optimal utilization of network links. Dual-Structure Multicast (DuSM) [22] leverages different forwarding structures for high-bandwidth and low-bandwidth flows. This improves scalability and link utilization of SDN-based data centers. Jia *et al.* [23] present a way to efficiently organize forwarding entries based on prime numbers and the Chinese remainder theorem. This reduces the required state in forwarding devices and allows more efficient implementation. In [24] the authors propose a SDN-based multicast switching system that leverages bloom filters to reduce the number of TCAM-entries.

2) RESILIENCE FOR TRADITIONAL MULTICAST

Shen *et al.* [25] modify Steiner trees to include recovery nodes in the multicast distribution tree. The recovery nodes cache IPMC traffic temporarily and resend it after reconvergence when the destination notified the recovery point because it did not get all packets due to a failure. The authors of [26] evaluate several algorithms that generate node-redundant multicast distribution trees. They analyse the number of forwarding entries and the effect of node failures. In [27] the authors propose to deploy primary and backup multicast trees in SDN networks. The header of multicast packets contains an ID that identifies the distribution tree on which the packet is forwarded. When a failure is detected, the controller reconfigures affected sources to send packets along a working backup tree. Pfeiffenberger *et al.* [28] propose a similar method. Each node that is part of a distribution tree is the root of a backup tree that does not contain the unreachable NH but all downstream destinations of the primary distribution tree. When a node cannot forward a packet, it reroutes the packet on a backup tree by switching an VLAN tag in the packet header.

B. BIER-BASED MULTICAST

In this subsection we discuss work directly related to BIER. First, we define our work in contrast to other implementations. Then, we describe evaluations and extensions for BIER.

1) IMPLEMENTATIONS

We started with an implementation of BIER for the software switch bmv2 using $P4_{14}$. The protoype was documented at high level in a 2-page demo paper [4]. We then developed BIER-FRR and implemented a prototype for BIER and BIER-FRR on the software switch bmv2 using the newer variant $P4_{16}$ in [2]. That work demonstrated that the P4 language is expressive enough to implement also complex forwarding mechanisms and introduced a hierarchical controller hierarchy to quickly trigger FRR actions. The study compared restoration times for various failure cases and protection schemes at light load conditions of a few packets per second. Throughput measurements were not conducted as the bmv2 software switch is only a 'tool for developing, testing and debugging P4 data planes' [5] with low throughput (900 Mb/s) [29] and not for application in real networks. In contrast, this paper shows that BIER and BIER FRR can be implemented also on high-performance P4-programmable hardware, i.e., the switching ASIC Tofino, which entails additional functional and runtime constraints for implementations to achieve high throughput. Experimental measurement studies in a 100 Gb/s hardware testbed reveal performance challenges due to recirculations. As this is a general problem for some P4 programs, we derive recommendations to cope with them and validate them in our hardware testbed.

We know only a single BIER implementation by other authors which is based on OpenFlow and presented in [30], [31]. Their approach suffers from two major shortcomings. First, the BIER bit string is encoded in a MPLS header which is the only way to encode arbitrary bit strings in OpenFlow. This limits the bit string length, and thus the number of receivers, to 20 which is the length of an MPLS label. Second, the implementation performs an exact match on the bitstring. If a subscriber changes, the match does not work anymore and a local BIER agent that is not part of the OpenFlow protocol needs to process the packet. Therefore, we consider this project only as an early BIER-based prototype for OpenFlow and not as a production-ready BIER implementation.

2) EVALUATIONS AND EXTENSIONS OF BIER-BASED MULTICAST

The authors of [32] perform a simulation-based evaluation of BIER. They find that on metrics like delivery ratios and retransmissions BIER performs as well as traditional IPMC but has better link usage and no per-flow or per-group state in core devices.

Eckert *et al.* [33] propose an extension for BIER that allows for traffic engineering (BIER-TE). In addition to the egress nodes, the BIER header encodes the distribution tree of a packet. In [34] the authors propose 1 + 1 protection for BIER-TE. The traffic is transported on two disjoint distribution trees, which delivers the traffic even if one tree is interrupted by a failure.

C. PACKET RECIRCULATION IN P4

Hauser *et al.* [35] show in their P4 survey that packet recirculation is not used only in this BIER implementation but also in other P4 projects. In [36] the authors implement a congestion control mechanism in P4 and leverage packet recirculation to create notification packets, update their header fields, and send them to appropriate monitoring nodes. The authors of [37] present a content-based publish/subscribe mechanism in P4 where they introduce a new header stack that requires packet recirculation for processing. Uddin *et al.* [38] implement multi-protocol edge switching for IoT based on P4. Packet recirculation is used to process packets a second time after they have been decrypted.

III. BIT INDEX EXPLICIT REPLICATION (BIER)

In this Section we explain BIER. First, we give an overview. Then we describe the BIER forwarding table and how BIER packets are processed. Afterwards, we show a forwarding example. Finally, we review tunnel-based BIER-FRR.

A. BIER OVERVIEW

First, we introduce the BIER domain. Then, we present the layered BIER architecture followed by the BIER header. Finally, we describe BIER forwarding.

1) BIER DOMAIN

Figure 2 shows the concept of the BIER domain. When bit-forwarding ingress routers (BFIRs) receive an IPMC packet they push a BIER header onto it and forward the packet into the BIER domain. The BIER header identifies all destinations of the BIER packet within the BIER domain, i.e., bit-forwarding egress routers (BFERs). Bit-forwarding routers (BFRs) forward the BIER packets to all BFERs indicated in its BIER header. Thereby, packets are replicated and

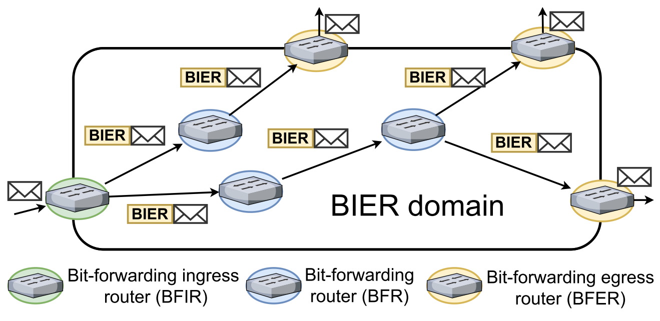

BIER
BIER
BIER
BIER
BIER
BIER
BIER domain
Bit-forwarding ingress router (BFIR)
Bit-forwarding router (BFR)
Bit-forwarding egress router (BFER)

Figure 2. The concept of the BIER domain [39].

forwarded to multiple next-hops (NHs) but only one packet is sent over any involved link. The paths towards the destinations are provided by the Interior Gateway Protocol (IGP), i.e., the routing underlay. Therefore, from a specific BFIR to a specific BFER, the BIER packet follows the same path as unicast traffic. Finally, BFERs remove the BIER header.

2) THE LAYERED BIER ARCHITECTURE

The BIER architecture consists of three components. The IPMC layer, the BIER layer and the routing underlay. Figure 3 shows the three layers, their composition, and interaction. The IPMC layer contains the sources and subscribers of IPMC traffic. The BIER layer acts as a transport layer for IPMC traffic. It consists of the BIER domain which is connected to the IPMC layer at the BFIRs, and BFERs. Therefore, the BIER layer acts as a point-to-multipoint tunnel from an IPMC source to multiple subscribers. The routing underlay refers to the IGP which provides the paths to all destinations within the network.

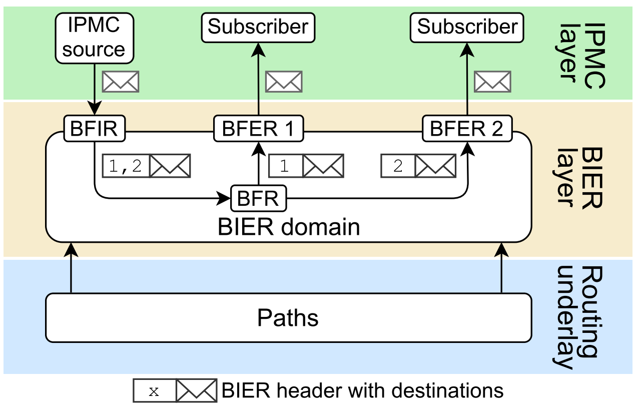


Figure 3. IPMC packets are transmitted over a layered BIER architecture; the paths are defined by the information from the routing underlay [39].

3) BIER HEADER

The BIER header contains a bit string to indicate the destinations of a BIER packet. To that end, each BFER is assigned an unique number that corresponds to a bit position in that bit string, starting by 1 for the least-significant bit. If a BFER should receive a copy of the IPMC packet, its bit is activated in the bit string in the BIER header of the packet. To facilitate readability we refer to the bit string in the BIER header of a BIER packet with the term 'BitString'.

4) BIER FORWARDING

A BFR forwards a packet copy to any neighbor over which at least one destination of the packet indicated by its BitString is reached according to the paths from the routing underlay. Before a packet is forwarded to a specific NH, the BFR clears all bits that correspond to BFERs that are reached via other NHs from the BitString of that packet. This prevents duplicates at the BFERs.

B. BIFT STRUCTURE

BFRs use the Bit Index Forwarding Table (BIFT) to determine the NHs of a BIER packet. Table 1 shows the BIFT of BFR 1 from Figure 4. For each BFER there is one entry in the BIFT. Entries of the BIFT consist of a NH, and a so-called F-BM. The F-BM is a bit string similar to the BitString. It records which BFERs have the same NH. In the F-BM of an BIFT entry the bits of BFERs are activated which are reached over the NH of that entry. Therefore, BFERs with the same NH have the same F-BM. BFRs use the F-BM to clear bits from the BitString of a packet before it is forwarded to a NH.

Table 1. BIFT of BFR 1 in the example of Figure 4 [39].

BFER	NH	F-BM
1	-	-
2	2	1010
3	3	0100
4	2	1010

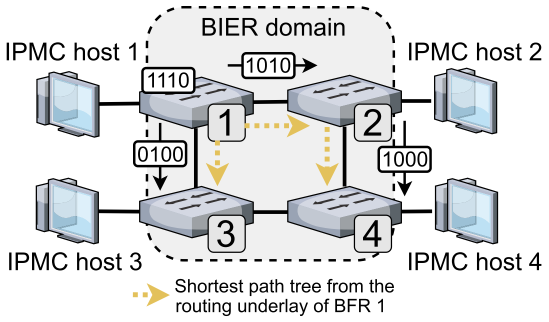


Figure 4. Example of a BIER topology and BitStrings of forwarded BIER packets [39].

C. BIER PACKET PROCESSING

When a BFR receives a BIER packet, it first stores the BitString of the packet in a separate bit string to account to which BFERs a packet has to be sent. In the following, we refer to that bit string with the term 'remaining bits'. The following procedure is repeated, until the remaining bits contain no activated bits anymore [1].

The BFR determines the least-significant activated bit in the remaining bits. The BFER that corresponds to that bit is used for a lookup in the BIFT. If a matching entry is found, it results in a NH *nh* and the F-BM *fbm* and the BFR creates a copy of the BIER packet. The BFR uses *fbm* to clear bits from the BitString of the packet copy. To that end, the BFR performs a bitwise AND operation of *fbm* and the BitString of the packet copy and writes the result into the BitString of the packet copy. This procedure is called applying the F-BM. It leaves only bits of BFERs in the BitString active that are reached over *nh*. The packet copy is then forwarded to *nh*. Afterwards, the bits of BFERs to which a packets has just been sent are cleared from the remaining bits. To that end, the BFR performs a bitwise AND operation of the bitwise complement of *fbm* with the remaining bits. The result is then stored in the remaining bits.

D. BIER FORWARDING EXAMPLE

Figure 4 shows a topology with four BIER devices where each is BFIR, BFR, and BFER. Table 1 shows the BIFT of BFR 1.

BFR 1 receives an IPMC packet from IPMC host 1 which should be distributed to all other IPMC hosts. Therefore, BFIR 1 pushes a BIER header with the BitString `1110` to the IPMC packet.

Then, BFR 1 determines the least-significant activated bit in the BIER header which corresponds to BFER 2. This BFER is used for lookup in the BIFT, which results in the F-BM `1010` and the NH BFR 2. BFR 1 creates a packet copy and applies the F-BM to its BitString. Then, the packet copy with the BitString `1010` is forwarded to BFR 2. Finally, the activated bits of the F-BM are cleared from the remaining bits which leaves the bit string `0100`.

This leaves only one bit active which identifies BFER 3. After the F-BM `0100` is applied to the BitString of a packet copy, it is forwarded to BFR 3 with the BitString `0100`. After clearing the bits of the F-BM from the remaining bits, processing stops because no active bits remain.

E. TUNNEL-BASED BIER-FRR

Tunnel-based BIER-FRR is used to deliver BIER traffic even when NHs are unreachable due to link or node failures. When a BFR detects that a NH is unreachable, e.g., by loss-of-carrier, loss-of-light, or a bidirectional forwarding detection (BFD[1]) [40] for BIER [41], it becomes the point of local repair (PLR) by tunneling the BIER packet through the routing underlay to nodes downstream in the BIER distribution tree. The tunnel may be affected by the failure, too. However, FRR mechanisms or timely updates of the FIB in the routing underlay restore connectivity for unicast traffic faster than for BIER traffic because recomputation of BIER entries can start only after the FIB of the routing underlay has been updated. Tunnel-based BIER-FRR can be configured either for link protection or node protection. BIER-FRR with link protection tunnels the BIER packet to the NH where the tunnel header is removed and the BIER header is processed again. BIER-FRR with node protection tunnels copies of the BIER packets to all next-next-hops (NNHs) in the distribution tree.

IV. INTRODUCTION TO P4

In this section we briefly review fundamentals of P4 [3]. First, we give an short overview of the P4 processing pipeline. Afterwards, we explain packet cloning and packet recirculation and point out important properties.

A. P4 PIPELINE

In this subsection we review the P4 processing pipeline. We explain its composition, transient and persistent memory, match + action tables, control blocks, packet cloning

[1]When a BFR is established between two nodes, they periodically exchange notifications about their status.

and packet recirculation. Figure 5 shows the concept of the P4 processing pipeline.

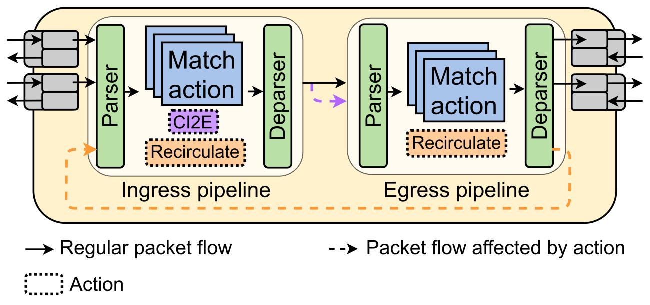


Figure 5. P4 processing pipeline.

1) COMPOSITION

The P4 pipeline consists of an ingress pipeline and an egress pipeline. They process packets in a similar fashion, i.e., both contain a parser, a match + action pipeline, and a deparser. When a packet arrives at the switch, it is first processed by the ingress pipeline. The header fields of the packet are parsed and carried along with the packet through the ingress pipeline. The parser is followed by a match + action pipeline which consists of a sequence of conditional statements, table matches, and primitive operations. Afterwards, the packet is deparsed and sent to the egress pipeline for further processing. Finally, the packet is sent through the specified egress port which has to be set in the ingress pipeline and cannot be changed in the egress pipeline.

The P4 program defines the parser and the deparser, which allows the use of custom packet headers. In addition, the P4 program describes the control flow of the match + action pipeline in the ingress pipeline and egress pipeline, respectively.

2) CONTROL BLOCKS

Both the ingress and egress pipeline can be divided into so-called control blocks for structuring. Control blocks are used to clearly separate functionality for different protocols like IP, BIER, and Ethernet, i.e., the IP control block contains Match + Action Tables (MATs) and operations that are applied only to IP packets, etc. In this paper we focus only on the BIER control block.

3) Match+Action TABLES (MATs)

MATs execute packet-dependent actions by matching packet header fields against MAT entries. To that end, an entry contains one or more match fields, and an action set. When a packet is matched against a MAT, the match fields of the entries are compared with specified header fields of the packet. An action set consists of one or more actions, e.g., reading or writing a header field, mathematical operations, setting the egress port of the packet, etc. It is not possible to match a packet on the same MAT multiple times.

B. PACKET CLONING

The operation clone-ingress-to-egress (CI2E) allows packet replication in P4. It can be called only in the ingress pipeline. At the end of the ingress pipeline, a copy of the packet is created. However, the packet copy resembles the packet that has been parsed in the beginning of the ingress pipeline, i.e., the header changes performed during processing in the ingress pipeline are reverted. This is illustrated in Figure 6.

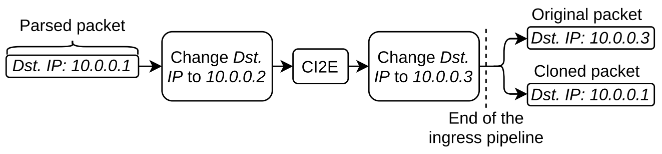


Figure 6. An example of the clone-ingress-to-egress (CI2E) operation [39].

If an egress port has been provided as a parameter, the egress port of the clone is set to that port. Both the original and cloned packet are processed independently in the egress pipeline. The cloned packet carries a flag to identify it as a clone.

C. PACKET RECIRCULATION

In this subsectin we explain the packet recirculation operation. First, we explain its working. Afterwards, we introduce the term recirculation capacity.

1) FUNCTIONALITY

P4 allows to recirculate a packet for processing it by the pipeline a second time. We use this feature to implement the iterative packet processing of BIER as described in Section III-C as P4 offers no other possibility to implement processing loops.

P4 leverages a switch-intern recirculation port for packet recirculation. When a packet should be recirculated, its egress port has to be set to the recirculation port during processing in the ingress pipeline. The flow of a packet through the pipeline when it is recirculated is shown in Figure 7. The packet is still processed by the entire processing pipeline, i.e., the ingress pipeline and egress pipeline. However, after the packet has been deparsed, it is not sent through a regular physical egress port but pushed back into the switch-intern recirculation port. The packet is then processed as if it has been received on a physical port. The recirculation port has the same capacity

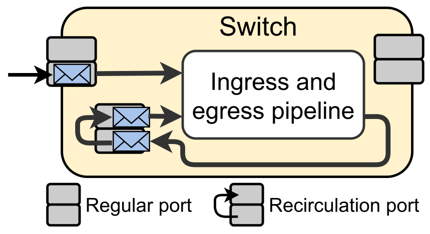


Figure 7. A packet is recirculated to a recirculation port and traverses the ingress and egress pipeline for a second time.

as the physical ports. For example, when two physical ports receive traffic at line rate and each packet is recirculated once, the recirculation port receives recirculated packets at double line rate, which causes packet loss.

2) RECIRCULATION CAPACITY

To discuss the effect of packet loss due to many recirculations we introduce the term 'recirculation capacity'. It is the available capacity to process recirculation traffic. Additional recirculation capacity is provided by using physical ports in loopback mode. When the forwarding device switches a packet to an egress port that is configured as a loopback port, the packet is immediately placed in the ingress of that port, instead. The packet is then processed as if it has been received on that port as usual, i.e., by the parser, the ingress and egress pipeline, and the deparser. Only traffic that has to be recirculated is switched to recirculation ports. In the following the term 'recirculation port' refers to a physical port in loopback mode, or the switch-intern recirculation port. When recirculation ports are required, the switch-intern recircution port should be used first, before any physical ports are configured as loopback ports. Only packets that are recirculated require recirculation capacity, i.e., common unicast traffic, e.g., as in regular IP unicast forwarding, is not recirculated, and therefore, does not occupy any recirculation capacity.

When multiple recirculation ports are deployed to increase the recirculation capacity, packets that should be recirculated need to be distributed over these ports. There are different distribution strategies. We developed a round-robin-based distribution approach for recirculation traffic to distribute the load equally over all recirculation ports. We store in a register which recirculation port receives the next packet which should be recirculated. When a packet has to be sent to a recirculation port, that register is accessed and updated in one atomic operation. This prevents any race conditions when traffic is distributed. Thus, this distribution strategy has two advantages. First, if n recirculation ports are used, the available recirculation capacity is increased to $n \cdot linerate$. Second, the equal distribution of recirculation traffic over all recirculation ports guarantees the full utilization of available recirculation capacities before packet loss occurs.

V. P4 IMPLEMENTATION OF BIER AND BIER-FRR FOR TOFINO

In this section we give an overview of the P4 implementation of BIER and tunnel-based BIER-FRR. First, we discuss the implementation basis. Afterwards, we give an overview of the processing of BIER packets, in particular we discuss packet recirculation.

A. CODEBASE

In [2] we presented a software-based prototype of a P4$_{16}$ implementation of BIER and tunnel-based BIER-FRR for the P4 software switch bmv2. We provided a very detailed description of the P4 programs including MATs with match fields and action parameters, control blocks, and applied operations. The prototype and the controller are publicly available on GitHub.[2]

In this paper we refrain from including a detailed technical description of the implementation for the Tofino. However, the source code[3] can be accessed by anyone on GitHub. In the following, we only explain important aspects of the hardware-based implementation to facilitate the understanding of the evaluation in Section VI and the model derivations in Section VII.

B. BIER PROCESSING

First, we describe the implementation of regular BIER forwarding on the Tofino. Afterwards, we explain operation of tunnel-based BIER-FRR.

1) BIER FORWARDING

Figure 8 shows how a BIER packet is processed once in the packet processing pipeline.

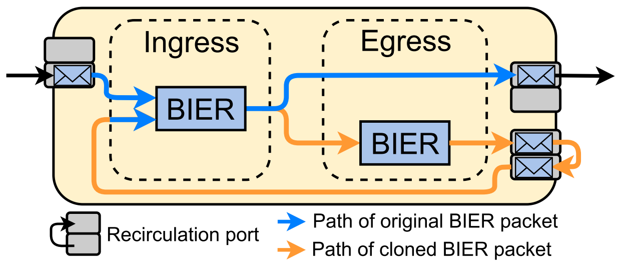


Figure 8. **Paket flow of a BIER packet in the processing pipeline.**

When the switch receives a BIER packet it is processed by the BIER control block. First, the BitString of the packet is matched against the BIFT which determines the egress port and the F-BM. The F-BM is applied to the BitString of the packet and cleared from the remaining bits. If the remaining bits still contain activated bits, CI2E is called and the egress port is set to a recirculation port so that the packet will be processed again. After the ingress pipeline, the copy is created and both packet instances enter the egress pipeline independently of each other. The original packet is sent through an egress port towards its NH. The packet clone is processed by a second BIER control block in the egress pipeline which sets the BitString of the packet copy to the remaining bits. Since the egress port of the packet clone is a recirculation port, the packet is recirculated, i.e., it is processed by the ingress pipeline again.

BIER forwarding removes BIER headers from packets that leave the BIER domain, and adds IP headers for tunneling through the routing underlay by tunnel-based BIER-FRR. Whenever a header is added or removed, the packet is recirculated for further processing.

When a BIER packet has more than one NH, two challenges appear. First, the BitString of a BIER packet has to be

[2]https://github.com/uni-tue-kn/p4-bier
[3]https://github.com/uni-tue-kn/p4-bier-tofino

matched several times against the BIFT to determine all NHs. However, matching a packet multiple times against the same MAT is not possible in P4. Second, multiple packet copies have to be created for forwarding. However, P4 does not allow to dynamically generate more than one copy of a packet. Therefore, we implemented a packet processing behavior where in each pipeline iteration one packet is forwarded to a NH and a copy of the packet is recirculated for further processing. This is repeated until all NHs receive a packet over which at least one destination of the BIER packet is reached. Figure 9 shows the processing of a BIER packet which has to be forwarded to three neighbors. In the first and second pipeline iteration the original BIER packet is sent through a physical egress port towards a NH and the copied BIER packet is recirculated by sending the packet copy to a recirculation port. In the last iteration when the remaining bits contain no activated bits anymore, no further packet copy is required and only the original BIER packet is sent through the egress port. In total, the packet needs to be recirculated two times to forward it to all three NHs. Therefore, in general, a BIER packet with n NHs, has to be recirculated $n-1$ times and the first NH can be served without packet recirculation.

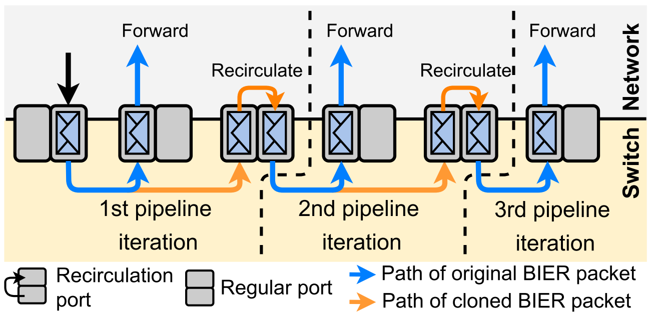


Figure 9. BIER processing over multiple pipeline iterations.

2) FORWARDING WITH TUNNEL-BASED BIER-FRR

The concept of tunnel-based BIER-FRR has been proposed in [2]. We implement it for the Tofino as follows.

The switch monitors the status of its ports as described in Section. When the match on the BIFT results in a NH which is reached by a port that is currently down, the processing of the BIER packet differs in the following way from the BIER processing described above. An IP header is added to the original BIER packet to tunnel the packet through the routing underlay towards an appropriate node in the BIER distribution tree. The egress port of the original packet is set to a recirculation port to process the IP header in another pipeline iteration, i.e., forward the IP packet to the right NH.

VI. PERFORMANCE EVALUATION OF THE P4-BASED HARDWARE PROTOTYPE

In this section we perform experiments to evaluate the performance of the P4-based hardware prototype for BIER regarding Layer-2 throughput and failover time, i.e., the time until BIER traffic is successfully delivered after a network failure.

A. FAILOVER TIME FOR BIER TRAFFIC

Here we evaluate the restoration time after a failure in three scenarios and vary the protection properties of IP and BIER. First, only the IP FIB and BIER FIB are updated by the controller, respectively, and no FRR mechanisms are activated. This process is triggered by a device that detects a failure. It notifies the controller which computes new forwarding rules and updates the IP and BIER FIB of affected devices. This scenario measures the time until the BIER FIB is updated after a failure, which is our baseline restoration time. The control plane, i.e., the controller, is directly connected to the P4 switch, which keeps the delay to a minimum in comparison to networks where the controller is several hops away.

Second, only BIER-FRR is deployed. In this scenario BIER is able to utilize tunnel-based BIER-FRR in case of a failure. However, FRR for IP traffic remains deactivated. Thus, IP traffic can be forwarded only after the IP FIB is updated.

Third, both IP-FRR and BIER-FRR are deployed. This scenario evaluates how quickly the P4 switch can react to network failures and restore connectivity of BIER and IP forwarding.

In the following, we first explain the setup and the metric. Then, we present our results. Finally, we discuss the influence of the setup on the results.

1) EXPERIMENT SETUP

Figure 10 shows the testbed. The Tofino [6], a P4-programmable switching ASIC, is at the core of the hardware testbed. We utilize a Tofino based Edgecore Wedge 100BF-32X [7] switch with 32 100 Gb/s ports. An EXFO FTB-1 Pro [42] 100 Gb/s traffic generator is connected to the Tofino to generate a data stream that is as precise as possible. Furthermore, we deploy two bmv2s that act as BFRs and BFERs. The traffic generator, the controller and two bmv2s are connected to the Tofino. The traffic generator sends IPMC traffic to the Tofino. The IPMC traffic has been subscribed only by bmv2-1. As long as the link between the Tofino and bmv2-1 works, the BIER packets are forwarded on the primary path. When the Tofino detects a failure, it notifies the

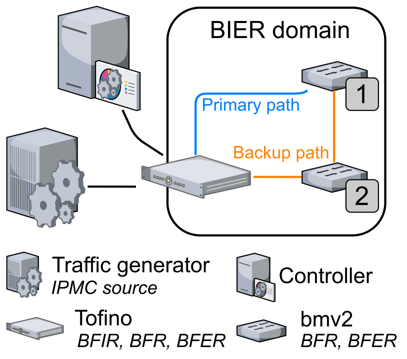


Figure 10. Experimental setup for evaluation of restoration time.

controller which computes new rules and updates forwarding entries of affected devices. In the meantime, the Tofino uses BIER-FRR to protect BIER traffic, and IP-FRR to protect IP traffic if enabled. This causes the Tofino to forward traffic on the backup path via bmv2-2 towards bmv2-1.

2) METRIC

We disable the link between the Tofino and bmv2-1 and measure the time until bmv2-1 receives BIER traffic again. We evaluate different combinations with and without IP-FRR and with and without BIER-FRR. To avoid congestion on the bmv2 and the VMs, the traffic generator sends only with 100 Mb/s, which has no impact on the results.

Figure 11 shows the average restoration time for the different deployed protection scenarios based on 10 runs which we discuss in the following. Confidence intervals are given on the base of a confidence level of 95%.

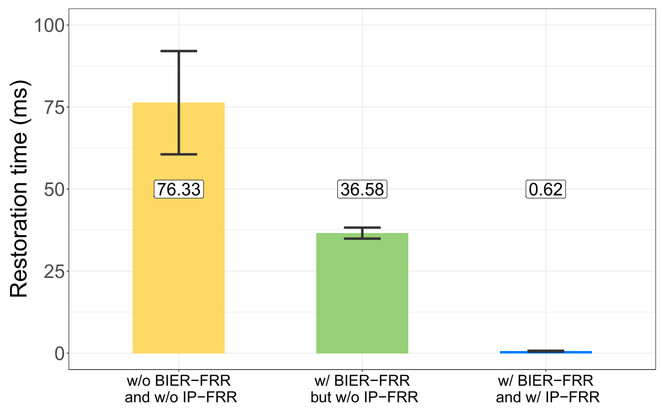


Figure 11. **Restoration time for BIER with different FRR strategies.**

3) FAILOVER TIME W/O BIER-FRR AND W/O IP-FRR

When no FRR mechanism is activated, multicast traffic arrives at the host only after the IP and BIER forwarding rules have been updated, which takes about 76 ms. The controller is directly connected to the Tofino. In a real deployment the controller may be multiple hops away, which would increase the restoration time significantly.

The same failover time is achieved without BIER-FRR but with IP-FRR, for which we do not present separate results. As BIER forwarding entries are updated only after IP forwarding entries have been updated, the use of IP-FRR in the network does not shorten the failover time for BIER traffic.

4) FAILOVER TIME W/BIER-FRR BUT W/O IP-FRR

When tunnel-based BIER-FRR but not IP-FRR is activated, bmv2-1 receives multicast traffic after 36 ms. In case of a failure, BIER-FRR tunnels the BIER traffic through the routing underlay. As soon as IP forwarding rules are updated, multicast traffic arrives at the host again. Since IP rules are updated faster than BIER rules, BIER-FRR decreases the restoration time for multicast traffic even if no IP-FRR mechanism is deployed.

5) FAILOVER TIME W/BIER-FRR AND W/IP-FRR

In the fastest and most resilient deployment both BIER-FRR and IP-FRR are activated. Then, multicast packets arrive at the host with virtually no delay after only 0.6 ms. In contrast to the previous scenario, unicast traffic is rerouted by IP-FRR which immediately restores connectivity for IP traffic.

6) INFLUENCE OF EXPERIMENTAL SETUP

The experimental setup (see Figure 10) features two BFERs on the base of bmv2 software switches with rather low performance compared to the Tofino-based hardware switch. However, we designed the experiment such that the low performance of these BFERs has no impact on results. bmv2 software switches can forward traffic with a rate up to 900 Mb/s [29]. By limiting the generated traffic rate to 100 Mb/s, the bmv2 switches forwarding and receiving BIER traffic are not overloaded so that bmv2-1 is able to measure correct restoration times. Furthermore, failure detection and protection switching are only carried out by the Tofino-based switch in the setup.

We now consider the impact of the hardware hosting the controller. When the controller is notified about a failure, it recomputes entries for IP and BIER forwarding tables. The computation time depends on the performance of the host and the size of the network in terms of number of nodes. Thus, the recomputation time may be significantly larger in larger networks, which increases the restoration time for BIER without any fast-reroute and for BIER with BIER-FRR but without IP-FRR. In contrast, the restoration time for BIER with BIER-FRR and IP-FRR is not impacted by the controller hardware or network size.

We discuss the impact of the signalling delay between the failure-detecting node and the controller. This delay was very low in our setup while it may be significantly larger in networks with large geographic extension or slow links. Such signalling delay adds to the restoration time for BIER without any fast-reroute and for BIER with BIER-FRR but without IP-FRR. The restoration time for BIER with BIER-FRR and IP-FRR is not impacted by that delay.

Finally, controller overload may occur when the controller needs to process too many messages, e.g., in case of a failure. This again has no impact on the restoration time for BIER with BIER-FRR and IP-FRR while it has significant impact on the restoration time for the other two settings.

B. THROUGHPUT FOR BIER TRAFFIC

The P4-based implementation of BIER described in Section V-B requires recirculation and is limited by the amount of recirculation capacity. The PSA defines a virtual port for this purpose. In this section we show the impact of insufficient recirculation capacity on throughput and the effect when additional physical recirculation ports, i.e., ports in loopback mode, are used for recirculation. We validate our experimental results in Section VI-C based on a theoretical model.

1) EXPERIMENTAL SETUP

The experimental setup is illustrated in Figure 12. A source node sends IPMC traffic to a BFIR. The BFIR encapsulates that traffic and sends it to a BFR. The BFR forwards the traffic to n BFERs which decapsulate the BIER traffic and send it as normal IPMC traffic to connected subscribers.

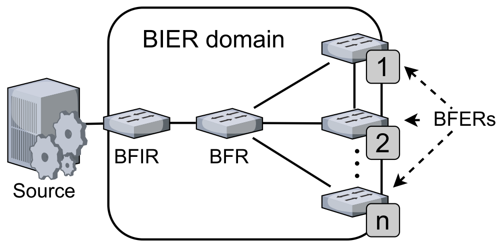


Figure 12. Theoretical setup for evaluation of BIER throughput.

The goal of the experiment is to evaluate the forwarding performance of the BFR depending on the number of NHs. With n NHs, BIER packets have to be recirculated $n-1$ times, and internal packet loss occurs if recirculation capacity does not suffice. The objective of the experiment is to measure the BIER throughput depending on the number of recirculation ports for which only physical loopback ports are utilized in the experiment. However, the n subscribers may see different throughput. The first BFER does not see any packet loss while the last BFER sees most packet loss. Therefore, we measure the rate of IPMC traffic received on Layer 2 at the last subscriber.

2) HARDWARE SETUP AND CONFIGURATION

Due to hardware restrictions in our lab, we utilize one traffic generator, one P4-capable hardware switch, and one server running multiple P4 software switches to build the logical setup sketched above. The hardware setup is shown in Figure 13. The traffic generator is the source of IPMC traffic and sends traffic to the BFIR. The traffic generator is also the subscriber of BFER n and measures the throughput of received IPMC traffic on Layer 2. The hardware switch acts as BFIR, BFR, and BFER n while BFERs 1 to $n-1$ are deployed as P4 software switches on the server. In addition, we collapse the BFIR and the BFR in the hardware switch so that packet forwarding from the BFIR to the BFR is not needed. Therefore, the traffic generator is the last NH of the BIER packet when it is processed by the BFR.

Packet recirculation is required after (1) encapsulation to enable further BIER processing, (2) decapsulation to enable further IP forwarding, and (3) BIER packet replication to enable BIER forwarding to additional NHs. We set up the hardware switch so that all recirculation operations in connection with encapsulation and decapsulation are supported by two dedicated ports in loopback mode and spend another k ports in loopback mode to support packet recirculation after packet replication. This models the competition for recirculation ports on a mere BFR as in the theoretical model.

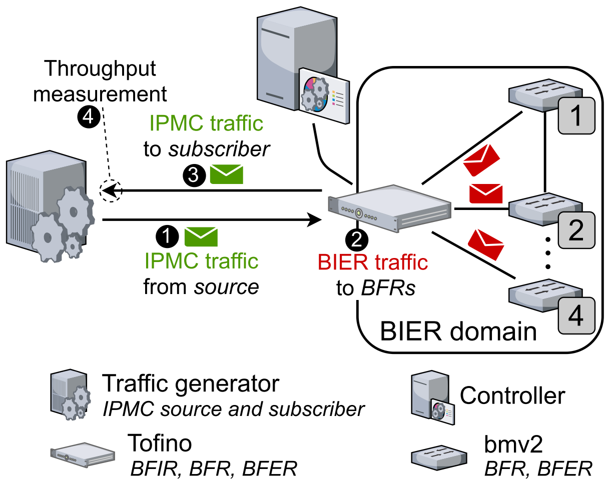


Figure 13. Hardware setup for evaluation of BIER throughput.

The P4 software switches are bmv2s that run alongside our controller on VMs on a server with an Intel Xeon Scalable Gold 6134 (8x 3.2 GHz) and 4 x 32 GB RAM. The P4 hardware switch is a Tofino [6] inside an Edgecore Wedge 100BF-32X [7] which is a 100 Gb/s P4-programmable switch with 32 ports. The traffic generator is an EXFO FTB-1 Pro [42] which generates up to 100 Gb/s. All devices are connected with QSFP28 cables which transmit up to 100 Gb/s.

3) INFLUENCE OF EXPERIMENTAL SETUP

The presented setup contains only a single Tofino-based switch which is partitioned and utilized as a single BFIR/BFR and a single BFER. All other BFERs in this setting are software switches that support only significantly lower bit rates (900 Mb/s [29]) than the Tofino-based switch (100 Gb/s). However, this has no impact on results because we measure the rate received by the single BFER implemented on the Tofino-based hardware. Furthermore, packet loss by the low-performance software switches does not reduce the generated traffic rate as this is configured as a constant rate on the generator.

4) BIER THROUGHPUT MEASUREMENTS DEPENDING ON RECIRCULATION PORTS

The traffic generator sends IPMC traffic at a rate of 100 Gb/s to the hardware switch, the hardware switch encapsulates the IPMC traffic, forwards BIER traffic iteratively n-1 times to bmv2s, recirculates the BIER packet to process the last activated header bit, decapsulates the traffic as BFER n, and returns it back to the traffic generator, which measures the received IPMC rate on Layer-2. We start measuring only after a 30 seconds initialization phase to avoid any influences from the startup phase. After 30 seconds, the traffic generator measures for 60 seconds the traffic arriving from the Tofino and reports the average Layer-2 throughput. We repeated experiments 10 times and computed confidence intervals with a confidence level of 95%. Their width was less than 0.5% of the measured average and, therefore, invisible.

Table 2. **Model predictions *T*(*i*) for BIER throughput and measured values *M*(*i*) (Gb/s); the latter are the same values as presented in Figure 14.**

Number of NHs:	1 NH		2 NHs		3 NHs		4 NHs	
Recirculation ports	$T(1)$	$M(1)$	$T(2)$	$M(2)$	$T(3)$	$M(3)$	$T(4)$	$M(4)$
1	100	99.32	100	99.32	38.2	43.3	15.74	19
2	100	99.32	100	99.32	100	99.32	53.14	50.5
3	100	99.32	100	99.32	100	99.32	100	99.32

Therefore, we omit them in future figures and tables for better readability.

In our experiments, we consider 1, 2, 3, and 4 NHs and utilize 1, 2, and 3 ports in loopback mode to support recirculation for BIER forwarding. The results are compiled in Figure 14.

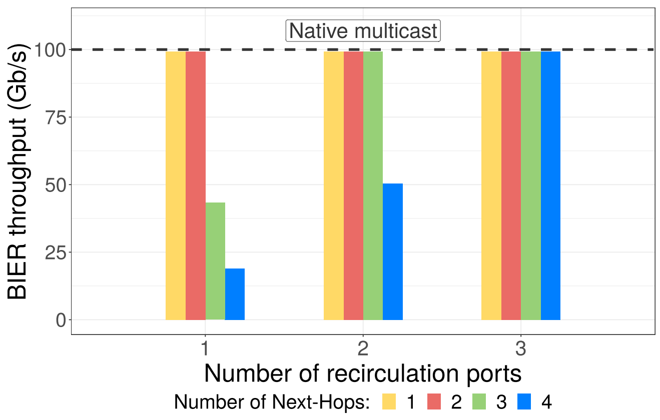


Figure 14. **Measured throughput of BIER and traditional IPMC on the 100 Gb/s Tofino-based switch for different numbers of NHs and recirculation ports.**

The left-most bar shows that with a single recirculation port, the last NH receives the full IPMC rate of 100 Gb/s if 1 NH is connected. The second bar from the left shows that the last NH still receives the full IPMC rate of 100 Gb/s if 2 NHs are connected. For 3 or 4 NHs, i.e., the third and fourth bar from the left, the IPMC traffic rate received by the last NH is reduced to 43 and 19 Gb/s, respectively.

With 2 recirculation ports, the last NH does not perceive a throughput degradation if at most 3 NHs, i.e., fifth to seventh bar from the left, are connected. For 4 NHs, i.e., eighth bar from the left, the IPMC traffic rate received by the last NH is reduced to 50 Gb/s.

And with 3 recirculation ports, even up to 4 NHs, i.e., ninth to twelfth bar from the left, can be supported without throughput degradation for the last NH.

Thus our experiments confirm that when multicast traffic arrives with 100 Gb/s at the Tofino, n-1 recirculation ports are needed to forward BIER traffic to n NHs without packet loss. This is different for a realistic multicast portion in the traffic mix, i.e., a minor fraction instead of 100%.

The hardware switch also supports traditional multicast in P4. With traditional multicast forwarding, all NHs receive 100 Gb/s regardless of the number of NHs. However, this comes with all the disadvantages of traditional IPMC we have discussed earlier.

C. THROUGHPUT MODEL FOR BIER FORWARDING WITH INSUFFICIENT RECIRCULATION CAPACITY

We model the throughput of BIER forwarding with insufficient recirculation capacity and validate the results with the experimentally measured values.

To forward a BIER packet to n NHs, it has to be recirculated $n-1$ times (see Section V-B). Any time a packet is sent to a recirculation, the packet is dropped with a certain probability if insufficient recirculation capacity is available. Due to the implemented round robin approach (see Section IV-C), the drop probability p is equal for all recirculation ports. The drop probability p in a system can be determined by comparing the available recirculation capacity and the sustainable recirculation load. The latter results from recirculations after BIER packet replication and takes packet loss into account. It is shown in the following formula.

$$C \cdot \sum_{m=1}^{n-1} (1-p)^m = k \cdot C \tag{1}$$

The available recirculation capacity is $k \cdot C$ where k is the number of recirculation ports and C is line capacity. The sustainable recirculation load is the sum of the successfully recirculated traffic rates after any number of recirculations. The traffic amount that has been successfully recirculated once is $C \cdot (1-p)$. The traffic amount that has been recirculated twice is $C \cdot (1-p)^2$, and so on. Therefore, the total amount is $C \cdot \sum_{m=1}^{n-1}(1-p)^m$.

We calculate the BIER throughput at any NH, i.e., after any number of recirculations. At the first NH, the throughput of the BIER traffic is C because the BIER packet is forwarded to the first NH before the packet is recirculated the first time. At the second NH, the BIER throughput is $C \cdot (1-p)$, at the third NH its $C \cdot (1-p)^2$, and so on. Therefore, the BIER throughput $T(i)$ at NH $1 \leq i \leq n$ is:

$$T(i) = C \cdot (1-p)^{i-1} \tag{2}$$

Table 2 shows the throughput predictions $T(i)$. We make predictions for the same scenarios as we evaluated in the performance evaluation in Section VI-B4 and compare them to the measured values $M(i)$.

The comparison shows that the model provides reasonable predictions for the BIER throughput.

VII. PROVISIONING RULE FOR RECIRCULATION PORTS

In this section we propose a provisioning rule for recirculation ports. It may be used for general P4-based applications requiring packet recirculation, not just for BIER forwarding. We first point out the importance for sufficient recirculation capacity. Then, we derive a general provisioning rule for recirculation ports and illustrate how their number depends on other factors. Finally, we apply that rule to provision the number of loopback ports for BFRs in the presence of traffic mixes.

A. IMPACT OF PACKET LOSS DUE TO MISSING RECIRCULATION CAPACITY

In Section IV-C we briefly discussed projects that leverage packet recirculation in P4. However, if recirculation capacity does not suffice and packets need to be recirculated several times, packet loss observed at the last stage may be quite high. We first illustrate this effect. If the packet loss probability due to missing recirculation capacity is p, then the overall packet loss probability after n recirculations is $p(n) = 1-(1-p)^n$. We illustrate this connection in Figure 15, which utilizes logarithmic scales to better view several orders of magnitude in packet loss. With only one recirculation, we obtain a diagonal for the overall packet loss. A fixed number of recirculations shifts the entire curve upwards, and with several recirculations like $n = 6$ or $n = 10$, the overall loss probability $p(6)$ or $p(10)$ is an order of magnitude larger than the packet loss probability p of a single recirculation step. Therefore, avoiding packet loss due to recirculations is important. Thus, sufficient recirculation capacity must be provisioned but overprovisioning is also costly since this means that entire ports at high speed cannot be utilized for operational traffic. Therefore, well-informed provisioning of recirculation ports is an important issue.

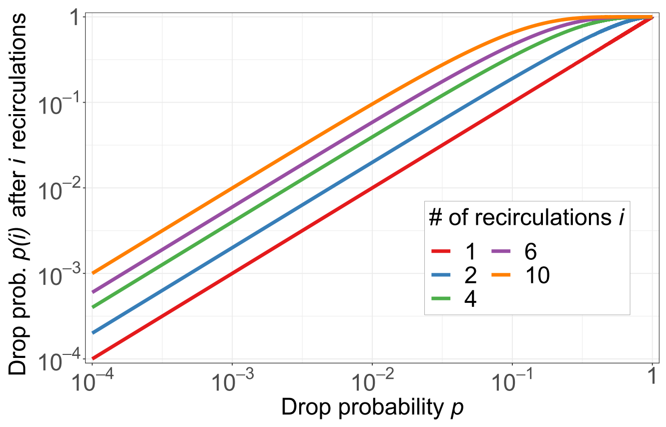


Figure 15. **Loss probability after multiple recirculations.**

B. DERIVATION OF A PROVISIONING RULE FOR RECIRCULATION PORTS

We first introduce the recirculation factor R and the utilization ratio U. Then, we use them to derive a provisioning rule for recirculation ports.

The recirculation factor R is the average number of recirculations per packet. Not all packets may be recirculated or the number how often a packet is recirculated depends on the particular packet.

The utilization ratio U describes the multiple by which a recirculation port can be higher utilized than a normal port. For example, if the average utilization of each normal port is 10%, then each recirculation port may be operated with a utilization of 40%, in particular if multiple of them are utilized. This corresponds to a utilization ratio of $U = 4$. We give some rationales for that idea. Normal ports at high speed are often underutilized in practice because bandwidths exist only in fixed granularities and usually link speeds are heavily overprovisioned to avoid upgrades in the near future. Furthermore, some links operate at lower utilization, others at higher utilization. Recirculation ports can be utilized to a higher degree. First, there is no need to keep the utilization of recirculation ports low for reasons like missing appropriate lower link speeds as it can be the case for normal ports. Second, recirculation ports are shared for all recirculation traffic of a switch so that resulting traffic fluctuations are lower and the utilization of the ports can be higher than the one of other ports.

If m incoming ports carry traffic with a recirculation factor R and a utilization ratio U can be used on the switch, then

$$m' = \left\lceil \frac{m \cdot R}{U} \right\rceil \tag{3}$$

describes the number of required recirculation ports.

C. ILLUSTRATION OF REQUIRED RECIRCULATION PORTS

For illustration purposes, we consider a P4 switch with 32 physical (external) ports and one virtual (internal) port in loopback mode for recirculations. If the capacity of that single virtual recirculation port does not suffice for recirculations, physical ports need to be turned into loopback mode as well and be used for recirculation. All recirculation ports are utilized in round-robin manner to ensure equal utilization among them.

Thus, the number of normal ports m plus the number of recirculation ports m' must be at most 33, i.e., 32 physical ports and 1 virtual port. Therefore, we find the smallest m' according to Equation 3, so that $m+m' \leq 33$ while maximizing m. The number of physical recirculation ports is $m'-1$ as the virtual port can also be used for recirculations. Figure 16 shows the number of physical recirculation ports depending on the recirculation factor R and the utilization ratio U. Since U depends on the specific use case and traffic mix, we present results for different values of U. Thereby, R and U are fractional numbers. While the number of recirculations for each packet is an integral number, the average number of recirculations per packet R is fractional. The number of physical recirculation ports increases with the recirculation factor R. Due to the fact that both m and m' are integers, the number of physical recirculation ports $(m - 1)$ is not monotonously increasing because for some R and U the sum

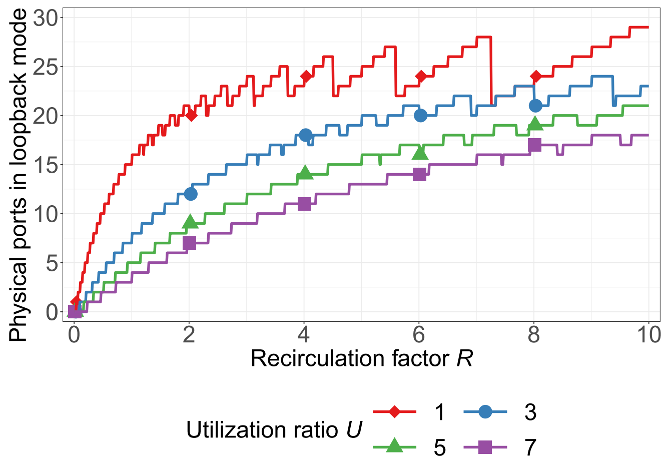


Figure 16. Number of physical ports in loopback mode.

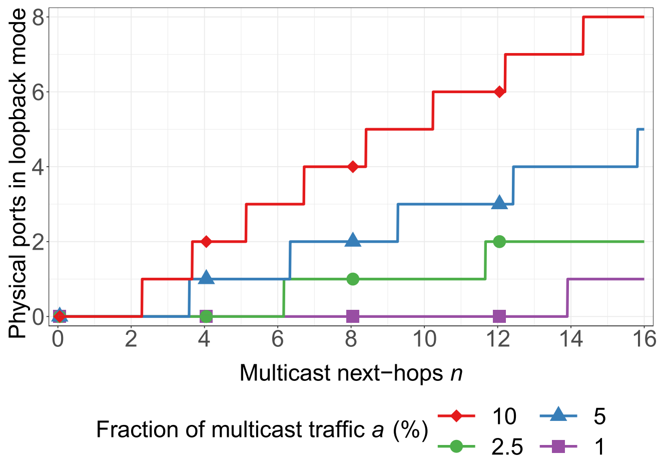


Figure 17. Physical ports in loopback mode for traffic mixes with realistic multicast portions.

$m + m'$ amounts to the maximum 33, and to lower values for other R and U.

The various curves show that the number of required physical recirculation ports decreases with increasing utilization ratio U. With a large recirculation factor $R \geq 3$ and a low utilization Ratio $U \leq 3$, half of the ports of the 32 port switch or even more need to be used for recirculation, which is expensive. However, with small $R < 1$ and large $U > 3$ the number of required physical recirculation ports is low because most of the traffic does not require packet recirculation, and due to the large utilization ratio U, the recirculation ports can cover significantly more traffic than normal ports. It is even possible that no physical recirculation port is needed if the recirculation capacity of the internal recirculation port can cover the recirculation load.

D. APPLICATION OF THE PROVISIONING METHOD TO TRAFFIC MIXES WITH BIER

In this section we make predictions for m', the number of recirculation ports, for traffic mixes with typical multicast portions. We assume different portions of multicast traffic $a \in \{0.01, 0.025, 0.05, 0.1\}$ and different average numbers of BIER NHs $n \in \{0, 2, 4, \ldots, 16\}$, i.e., each BIER packet is recirculated $n - 1$ times on average. Since unicast traffic is normally processed without recirculation, it does not need any recirculation capacity, i.e., its amount has no influence on the number of required recirculation ports and is, therefore, not considered in this analysis. Then, we calculate $R = a \cdot (n - 1)$, and assume $U = 4$. Again, we calculate the smallest m', i.e., like in Equation 3, so that $m + m' \leq 33$ while maximizing m. Figure 17 shows the number of physical recirculation ports depending on the average number of multicast NHs n and the fraction of multicast traffic a. If the fraction of multicast traffic is low like 1%, the capacity of the internal port suffices to serve up to 13 NHs on average. Moderate fractions of 2.5% multicast traffic require no physical recirculation port for up to 5 NHs, 1 physical recirculation port for up to 11 NHs, and 2 physical recirculation ports for 12 and more NHs. With 5% multicast traffic, the number of required physical recirculation ports increases almost linearly from zero to 5 with an increasing number of NHs. Large fractions of multicast traffic, like 10%, require up to 8 recirculation ports if the number of NHs is also large like 16. Under such conditions, 25% of the physical ports cannot be used for normal traffic forwarding as they are turned into loopback mode. However, the assumptions seem rather unlikely as multicast traffic typically makes up only a small proportion of the traffic.

VIII. CONCLUSION

The scalability of traditional IPMC is limited because core devices need to maintain IPMC group-dependent forwarding state and process lots of control traffic whenever topology or subscriptions change. Therefore, BIER has been introduced by the IETF as an efficient transport mechanism for IPMC traffic. State in BIER core devices does not depend on IPMC groups, and control traffic is only sent to border nodes, which increases scalability in comparison to traditional IPMC significantly. In addition, there are fast-reroute (FRR) mechanisms for BIER to minimize the effect of network failures. However, BIER cannot be configured on legacy devices as it implements a new protocol with a complex forwarding behavior.

In this paper we demonstrated a P4-based implementation of BIER with tunnel-based BIER-FRR, IP unicast with FRR, IP multicast, and Ethernet forwarding. The target platform is the P4-programmable switching ASIC Tofino which is used in the Edgecore Wedge 100BF-32X, a 32 100 Gb/s port high-performance P4 switch.

In an experimental study, we showed that BIER-FRR significantly reduces the restoration time after a failure, and in combination with IP-FRR, the restoration time is reduced to less than 1 ms. We confirmed that the prototype is able to forward traffic at a speed up to 100 Gb/s. However, under some conditions, less throughput is achieved when switch-internal recirculation ports are overloaded. As a remedy, we added more recirculation capacity by turning physical ports into loopback mode. We modelled BIER forwarding, predicted limited throughput due to missing recirculation capacity, and validated the results by measured values. Furthermore, we proposed a simple method for provisioning of physical recirculation ports. The approach was motivated by BIER, but holds for general P4 programs requiring recirculations. In a case study, we applied it to BIER with different mixes of unicast and multicast traffic and showed that only a few physical recirculation ports suffice under realistic conditions.

REFERENCES

[1] I. Wijnands. (Nov. 2017). *RFC 8279: Multicast Using Bit Index Explicit Replication (BIER)*. [Online]. Available: https://datatracker.ietf.org/doc/rfc8279/

[2] D. Merling, S. Lindner, and M. Menth, "P4-based implementation of BIER and BIER-FRR for scalable and resilient multicast," *J. Netw. Comput. Appl.*, vol. 169, Nov. 2020, Art. no. 102764.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, and J. Rexford, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

[4] W. Braun, J. Hartmann, and M. Menth, "Scalable and reliable software-defined multicast with BIER and P4," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, May 2017, pp. 905–906.

[5] P4lang. (2021). *behavioral-Model*. Accessed: Jan. 28, 2021. [Online]. Available: https://github.com/p4lang/behavioral-model

[6] Edge-Core Networks. (2017). *The World's Fastest & Most Programmable Networks*. [Online]. Available: https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/

[7] Edge-Core Networks. (2019). *Wedge100BF-32X/65X Switch*. [Online]. Available: https://www.edge-core.com/_upload/images/Wedge100BF-32X_65X_DS_R05_2019%1210.pdf

[8] M. Shahbaz, L. Suresh, J. Rexford, N. Feamster, O. Rottenstreich, and M. Hira, "Elmo: Source routed multicast for public clouds," in *Proc. ACM Special Interest Group Data Commun.*, 2019, pp. 458–471.

[9] S. Islam, N. Muslim, and J. W. Atwood, "A survey on multicasting in software-defined networking," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 1, pp. 355–387, 1st Quart., 2018.

[10] Z. AlSaeed, I. Ahmad, and I. Hussain, "Multicasting in software defined networks: A comprehensive survey," *J. Netw. Comput. Appl.*, vol. 104, pp. 61–77, Feb. 2018.

[11] J. Ráckert, J. Blendin, and D. Hausheer, "Software-defined multicast for over-the-top and overlay-based live streaming in ISP networks," *J. Netw. Syst. Manage.*, vol. 23, no. 2, pp. 280–308, Apr. 2015.

[12] J. Ruckert, J. Blendin, R. Hark, and D. Hausheer, "Flexible, efficient, and scalable software-defined over-the-top multicast for ISP environments with DynSdm," *IEEE Trans. Netw. Service Manage.*, vol. 13, no. 4, pp. 754–767, Dec. 2016.

[13] T. Humernbrum, B. Hagedorn, and S. Gorlatch, "Towards efficient multicast communication in software-defined networks," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, Jun. 2016, pp. 106–113.

[14] C. A. S. Oliveira, "Steiner trees and multicast," *Math. Aspects Netw. Routing Optim.*, vol. 53, pp. 29–45, Dec. 2011.

[15] L.-H. Huang, H.-J. Hung, C.-C. Lin, and D.-N. Yang, "Scalable and bandwidth-efficient multicast for software-defined networks," in *Proc. IEEE Global Commun. Conf.*, Dec. 2014, pp. 1890–1896.

[16] Z. Hu, D. Guo, J. Xie, and B. Ren, "Multicast routing with uncertain sources in software-defined network," in *Proc. IEEE/ACM 24th Int. Symp. Qual. Service (IWQoS)*, Jun. 2016, pp. 1–6.

[17] S. Zhou, H. Wang, S. Yi, and F. Zhu, "Cost-efficient and scalable multicast tree in software defined networking," in *Proc. Conf. Algorithms Archit. Parallel Process.*, 2015, pp. 562–605.

[18] J.-R. Jiang and S.-Y. Chen, "Constructing multiple Steiner trees for software-defined networking multicast," in *Proc. 11th Int. Conf. Future Internet Technol.*, Jun. 2016, pp. 1–6.

[19] B. Ren, D. Guo, J. Xie, W. Li, B. Yuan, and Y. Liu, "The packing problem of uncertain multicasts," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 16, p. e3985, Aug. 2017.

[20] Y.-D. Lin, Y.-C. Lai, H.-Y. Teng, C.-C. Liao, and Y.-C. Kao, "Scalable multicasting with multiple shared trees in software defined networking," *J. Netw. Comput. Appl.*, vol. 78, pp. 125–133, Jan. 2017.

[21] A. Iyer, P. Kumar, and V. Mann, "Avalanche: Data center multicast using software defined networking," in *Proc. 6th Int. Conf. Commun. Syst. Netw. (COMSNETS)*, Jan. 2014, pp. 1–8.

[22] W. Cui and C. Qian, "Scalable and load-balanced data center multicast," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2014, pp. 1–6.

[23] W.-K. Jia and L.-C. Wang, "A unified unicast and multicast routing and forwarding algorithm for software-defined datacenter networks," *IEEE J. Sel. Areas Commun.*, vol. 31, no. 12, pp. 2646–2657, Dec. 2013.

[24] M. J. Reed, M. Al-Naday, N. Thomos, D. Trossen, and G. Petropoulos, "Stateless multicast switching in software defined networks," in *Proc. IEEE Int. Conf. Commun.*, May 2016, pp. 1–7.

[25] S.-H. Shen, L.-H. Huang, D.-N. Yang, and W.-T. Chen, "Reliable multicast routing for software-defined networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 181–189.

[26] M. Popovic, R. Khalili, and J.-Y. Le Boudec, "Performance comparison of node-redundant multicast distribution trees in SDN networks," in *Proc. Int. Conf. Networked Syst. (NetSys)*, Mar. 2017, pp. 1–8.

[27] D. Kotani, K. Suzuki, and H. Shimonishi, "A multicast tree management method supporting fast failure recovery and dynamic group membership changes in OpenFlow networks," *J. Inf. Process.*, vol. 24, no. 2, pp. 395–406, 2016.

[28] T. Pfeiffenberger, J. L. Du, P. B. Arruda, and A. Anzaloni, "Reliable and flexible communications for power systems: Fault-tolerant multicast with SDN/OpenFlow," in *Proc. 7th Int. Conf. New Technol., Mobility Secur. (NTMS)*, Jul. 2015, pp. 1–6.

[29] A. Bas. (Jan. 2018). *BMv2 Throughput*. [Online]. Available: https://github.com/p4lang/behavioral-model/issues/537#issuecomment-360537441

[30] A. Giorgetti, A. Sgambelluri, F. Paolucci, P. Castoldi, and F. Cugini, "First demonstration of SDN-based bit index explicit replication (BIER) multicasting," in *Proc. Eur. Conf. Netw. Commun. (EuCNC)*, Jun. 2017, pp. 1–6.

[31] A. Giorgetti, A. Sgambelluri, F. Paolucci, N. Sambo, P. Castoldi, and F. Cugini, "Bit index explicit replication (BIER) multicasting in transport networks," in *Proc. Int. Conf. Opt. Netw. Design Modeling (ONDM)*, May 2017, pp. 1–5.

[32] Y. Desmouceaux and T. Clausen, "Reliable multicast with BIER," *J. Commun. Netw.*, vol. 20, pp. 182–197, May 2018.

[33] T. Eckert. (Nov. 2017). *Traffic Engineering for Bit Index Explicit Replication BIER-TE*. [Online]. Available: http://tools.ietf.org/html/draft-eckert-bier-te-arch

[34] W. Braun, M. Albert, T. Eckert, and M. Menth, "Performance comparison of resilience mechanisms for stateless multicast using BIER," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, May 2017, pp. 230–238.

[35] F. Hauser, M. Haeberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with P4: Fundamentals, advances, and applied research," 2021, *arXiv:2101.10632*. [Online]. Available: https://arxiv.org/abs/2101.10632

[36] J. Geng, J. Yan, and Y. Zhang, "P4QCN: Congestion control using P4-capable device in data center networks," *Electronics*, vol. 8, p. 280, Mar. 2019.

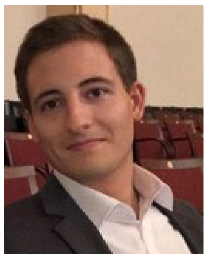
[37] C. Wernecke, H. Parzyjegla, G. Muhl, P. Danielis, and D. Timmermann, "Realizing content-based publish/subscribe with P4," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2018, pp. 1–7.

[38] M. Uddin, S. Mukherjee, H. Chang, and T. V. Lakshman, "SDN-based multi-protocol edge switching for IoT service automation," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 12, pp. 2775–2786, Dec. 2018.

[39] *Reprinted from Journal of Network and Computer Applications, vol. 169, Daniel Merling, Steffen Lindner, Michael Menth, P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast*, Elsevier, Amsterdam, The Netherlands, 2020.

[40] D. Katz. (Jul. 2004). *Bidirectional Forwarding Detection (BFD)*. [Online]. Available: https://datatracker.ietf.org/doc/rfc5880/
[41] Q. Xiong. (Oct. 2017). *BIER BFD*. [Online]. Available: https://datatracker.ietf.org/doc/draft-hu-bier-bfd/
[42] EXFO. (2019). *FTB-1v2/FTB-1 Pro Platform*. [Online]. Available: https://www.exfo.com/umbraco/surface/file/download/?ni=10900&cn=en-US&pi=5404

DANIEL MERLING received the master's degree from the Chair of Communication Networks of Prof. Dr. habil. Michael Menth, Eberhard Karls University, Tübingen, Germany, in 2017, where he is currently pursuing the Ph.D. degree. His research interests include software-defined networking, scalability, P4, routing and resilience issues, multicast, and congestion management.



STEFFEN LINDNER received the bachelor's and master's degrees from the Chair of Communication Networks of Prof. Dr. habil. Michael Menth. He is currently pursuing the Ph.D. degree with the Communication Networks Research Group, Eberhard Karls University, Tübingen, Germany. His research interests include software-defined networking, P4, and congestion management.

MICHAEL MENTH (Senior Member, IEEE) received the Diploma degree from The University of Texas at Austin, in 1998, the Ph.D. degree from the University of Ulm, Germany, in 2004, and the Habilitation degree from the University of Wuerzburg, Germany, in 2010. He is currently a Professor with the Department of Computer Science, University of Tuebingen, Germany, since 2010, and the Chair Holder of communication networks. His research interests include performance analysis and optimization of communication networks, resilience and routing issues, resource and congestion management, industrial networking and the Internet of Things, software-defined networking, and the Internet protocols.

1.3 Alternative Best Effort (ABE) for Service Differentiation: Trading Loss versus Delay

https://doi.org/10.1109/TNET.2022.3221553

Alternative Best Effort (ABE) for Service Differentiation: Trading Loss versus Delay

Steffen Lindner, Gabriel Paradzik, Michael Menth
Chair of Communication Networks, University of Tuebingen, Tuebingen, Germany
{steffen.lindner, gabriel.paradzik, menth}@uni-tuebingen.de

***Abstract*—The idea of an Alternative Best Effort (ABE) per-hop behaviour (PHB) emerged about 20 years ago. It provides a low-delay traffic class in the Internet at the expense of more packet loss than Best Effort (BE). Therefore, ABE is better suited than BE for loss-tolerant but delay-sensitive applications. Furthermore, ABE traffic should not degrade the service for BE traffic in terms of packet loss and delay. Therefore, Internet service providers may leave the choice of using BE or ABE to their customers as they achieve service differentiation without compromising other traffic.**

In this work, we revisit ABE and pursue the fundamental question whether an ABE service is technically feasible, how its service would look like and interact with existing transport protocols? We present a novel scheduler called Deadlines, Saved Credits, and Decay (DSCD) for combined scheduling of BE and ABE traffic. It allows to control ABE's delay advantage over BE and copes with varying bandwidth. We provide an implementation of DSCD in the Linux network stack and demonstrate its efficiency. A side product of the implementation is an efficient approximation of the exponential function in the kernel and a bandwidth estimation method that even works at moderate link utilization. We study DSCD in a semi-virtualized testbed with real networking stacks to understand implications for transport protocols in a BE/ABE Internet. The study analyzes ABE's impact on loss and delay under various conditions and gives recommendations for configuration.

***Index Terms*—Internet Protocol, traffic classes, service differentiation, packet scheduling, Alternative Best Effort (ABE), inter-class fairness**

I. Introduction

Over the last decades, increasing buffer sizes in network devices have led to temporarily long end-to-end delays. This phenomenon is known as bufferbloat [1]. Realtime applications, such as online gaming, video conference systems, or voice over IP applications, rely on small end-to-end delays. Table I gives examples.

TABLE I
Realtime applications with requirements regarding end-to-end delay and packet loss.

Application	E2E delay	Packet loss	Source
Online gaming (FPS)	20 ms – 80 ms	$\leq$ 5%	[2] [3]
Cloud gaming	$\leq$ 50 ms	$\leq$ 5%	[4] [5] [6]
Voice over IP (VoIP)	$\leq$ 150 ms	1% – 3%	[7]

A common solution to this problem is service differentiation, where delay sensitive traffic is prioritized over other traffic. The differentiated services framework (DiffServ) [8] allows for service differentiation in IP networks, offering various per-hop behaviours (PHBs) which can be considered as traffic classes. An example is Expedited Forwarding (EF), where EF traffic is strictly preferred over other traffic classes [9]. However, Internet service providers (ISPs) generally do not leave the choice of the PHBs to their customers because EF traffic may impede BE traffic of other users. If EF traffic accounts for only a small fraction of a link's overall traffic, it is likely to encounter only little loss and delay. In the absence of financial incentives, users possibly send too much EF traffic so that loss and delay objectives of EF traffic may not be met.

In this light, we revisit the idea of an Alternative Best Effort (ABE) service class [10] which has been first proposed in 2000. It causes less packet delay at the expense of increased packet loss but does not degrade treatment of BE traffic with respect to loss and delay. ABE constitutes a PHB that may be particularly attractive for service differentiation in the Internet where end users may choose ABE for low-delay traffic without negatively impacting BE traffic. Furthermore, it may be attractive when strict prioritization, such as the EF PHB, is discussed controversial in the context of network neutrality [11] as ABE does not impact the service for BE traffic.

Various scheduling algorithms [12], [13] have been presented in the past that may be apt to support an ABE PHB, but they suffer from various shortcomings. They are complex so that they have been implemented only in simulations. They require that the link bandwidth is stable and known. While thoroughly investigated by simulations, they did not address the inherent problem that very low ABE traffic rates may lead to unacceptable high packet loss. Supporting a loss versus delay trade-off has also been discussed in the IETF but implementation details were not in scope [14].

The contribution of this work is manifold. We address the fundamental question whether an ABE service class is technically feasible, how it behaves with up to date transport protocols, and whether it can be implemented on modern hardware. To that end, we propose a novel scheduler, named Deadlines, Saved Credits, and Decay (DSCD), for combined scheduling of BE and ABE traffic and implement it in the Linux network stack. It requires the knowledge of the link bandwidth only for secondary tasks and copes with unknown or variable transmission capacity through continuous bandwidth estimation. DSCD is designed to avoid excessive packet loss for low ABE traffic rates as this is most detrimental even to realtime applications.

We investigate the performance of ABE in terms of loss and delay. We show the impact of configuration parameters, traffic types, and the ABE traffic rate on these metrics. We perform experiments with TCP variants and study both inter-protocol and inter-class fairness when traffic is carried over BE and ABE. The proposed bandwidth estimation method is fast, accurate, and efficient so that it also works at moderate link utilization. For its implementation, we developed a fast approximation of the exponential function in the kernel which may be reused for other purposes.

This paper is structured as follows. In Section II, we review related work. Section III presents the novel DSCD algorithm for joint scheduling of BE and ABE traffic. Afterwards, we point out relevant implementation details of DSCD in the Linux network stack in Section IV. In Section V we investigate the performance of BE and ABE traffic when scheduled with the DSCD scheduler in various networking scenarios. Finally, Section VI summarizes this work and Section VII draws conclusions.

II. Related Work

We review work in the context of traffic prioritization related to DSCD. First, we discuss the ABE schedulers *Duplicate Scheduling with Deadlines* (DSD) and *Delay Segment FIFO* (DSF). Then, we present AQM implementations in the Linux network stack and further traffic differentiation mechanisms.

A. Alternative Best Effort (ABE)

The authors of [12] present the concept of Alternative Best Effort (ABE) as a traffic class similar to BE. ABE provides a bounded-delay service class (green) and a Best Effort (BE) class (blue). Blue traffic achieves the same throughput as in a conventional FIFO system. Green traffic receives priority service whenever possible without harming blue traffic. The concept has also been presented in the IETF [10].

The authors subsequently present *Duplicate Scheduling with Deadlines* (DSD) as a combined scheduler for BE and ABE traffic. DSD utilizes separate, physical FIFO queues for blue and green packets. In addition, it leverages a virtual queue to simulate the queue behaviour of a single FIFO queue that serves both blue and green packets. Based on the fill state of the virtual queue the transmission time for blue packets in a FIFO system is computed, for which the link capacity must be known and stable. This FIFO system transmission time is taken as a deadline for the blue packets in the physical queue. The deadline for green packets in the physical queue is their arrival time plus a maximum tolerable delay. The basic dequeue procedure is as follows. If a blue packet needs to be sent to keep its deadline, it is dequeued and sent. If a green packet passed its deadline, it is dropped. With DSCD, green packets are sent whenever blue packets do not need to be sent. This allows green packets to overtake blue packets when previous green packets were dropped, which effects a delay advantage. DSD is similar to DSCD but utilizes an algorithm to share capacity between both physical queues in a sophisticated way whenever blue or green packets are eligible for transmission. However, the objective for this capacity sharing is debatable as it strives for per-class fairness. As a result, the associated algorithm is complex. The delay, throughput, and packet loss must be tracked so that the algorithm is hard to implement on real forwarding nodes. The performance of DSD has been evaluated by simulations.

B. Delay Segment FIFO (DSF)

Karsten et al. [13] consider multiple traffic classes i with class-specific delay targets D_i. They suggest Delay Segment FIFO (DSF) as an algorithm for scheduling packets such that the queuing delay of the packets is at most their class-specific delay target and at most the delay experienced in a FIFO queue. If these constraints cannot be met, packets are dropped. They use a physical packet queue for each traffic class and a joint slot queue for storing slots that contain the right to send a certain amount of bytes. The slot queue is partitioned into class-specific segments such that the overall capacity of the segments corresponding to the c most stringent traffic classes is $D_c \cdot C$. Thereby C is the link bandwidth which must be known a priori and stable. When a packet arrives, it is added with a deadline to the corresponding packet queue. Moreover, a slot is added to slot queue in a free segment belonging to the packet class of the packet or better. The algorithm is complex and uses a minimal throughput interference index to guarantee some kind of TCP fairness. The authors implement and evaluate DSF for the network simulator ns-3.

C. AQM Implementations in the Linux Network Stack

FQ-CoDel [15] is a packet scheduler and Active Queue Management (AQM) algorithm developed to mitigate the bufferbloat problem. It is based on deficit round robin (DRR) and CoDel and distinguishes between sparse and non-sparse flows. FQ-CoDel stochastically enqueues incoming packets based on their 5-tuple hash into different queues. Each queue is managed by the CoDel AQM. FQ-CoDel is the default queueing discipline in many Linux distributions.

Ramakrishnan et al. [16] present an implementation of FQ-PIE, a flow-based variation of Proportional Integral controller Enhanced (PIE), for the Linux network stack. They compare it to PIE and FQ-CoDel and evaluate the fairness among responsive and unresponsive flows. Further, they evaluate the fairness between different TCP versions, i.e., TCP Cubic and TCP BBR.

CAKE [17] is a network queue managment system designed for the home gateway. It includes bandwidth shaping, queue management, DiffServ handling and TCP ACK filtering. Further, it provides host and flow isolation. CAKE is part of the mainline Linux kernel and was developed for the OpenWrt router firmware.

D. Further Traffic Differentiation Mechanisms

RD [18] proposes two service classes: a high-transmission class and a low queuing delay class. Both classes are implemented by separate FIFO queues on a router. The next transmitted packet is selected according to the intended throughput ratio between both classes. To guarantee delay differentiation, the corresponding queue sizes are dynamical calculated. Queue sizes and service rates have to be calculated, which requires knowledge of the link capacity.

QJump [19] classifies different latency-sensitive levels. Packets from higher classes are rate-limited but can "jump-the-queue" over packets from lower classes. To provide rate-limitations and some kind of throughput fairness, QJump needs knowledge about the number of network nodes and link speeds. QJump was designed for datacenter applications. The authors evaluate QJump with simulations and a small real-world deployment.

Briscoe et al. [20] give a broad overview of techniques to reduce Internet latency. They categorize latency sources, e.g., caused by too large network buffers, and present their advantages and disadvantages.

Although FQ-CoDel, FQ-PIE, Cake, RD, and other mechanisms provide means for service differentiation, i.e., they enable low-delay forwarding for real-time or low-bitrate traffic, they do so at the expense of BE traffic or large flows. In contrast, the objective of DSCD is that BE traffic is not treated worse than in a pure FIFO system. Further, DSCD could be used in conjunction with existing AQMs such as FQ-CoDel.

III. Scheduling with Deadlines, Saved Credits, and Decay (DSCD)

We give an overview of DSCD's design idea and present its algorithm in detail.

A. Design Idea

Figure 1 illustrates the data structure of DSCD that we introduce incrementally. The DSCD scheduler utilizes two FIFO queues: one to enqueue BE traffic ($Q[BE]$) and one to enqueue ABE traffic ($Q[ABE]$). When a packet arrives, it is enqueued into the corresponding class-specific queue based on its DiffServ code point (DSCP). In addition, ABE packets are equipped with a deadline at enqueue, which is the enqueue time plus the delay threshold T_d. When a non-empty BE queue is served, a packet is removed and dequeued for forwarding. When a non-empty ABE queue is served, packets are removed. They are dequeued if their deadline is met, otherwise they are dropped. The unused transmission capacity may be used by subsequent ABE packets, either immediately or later, so that they can take over BE packets without delaying them.

DSCD achieves this behavior by dequeuing packets from the BE and ABE packet queue using so-called credits. To that end, DSCD maintains a FIFO queue Qc for credit elements. Whenever a packet arrives, a credit element with the packet's size and traffic class is inserted into the credit queue ❶. The credit is needed for dequeuing packets. For that purpose, two class-specific credit counters are maintained ($CC[ABE]$, $CC[BE]$). The first packet of a non-empty queue can be dequeued only if the corresponding credit counter is at least the packet's size. If so, the packet can be removed from its queue. If the packet belongs to the ABE class and its deadline has passed, the packet is dropped. Otherwise, the corresponding credit counter is decremented by the packet's size and the packet can be forwarded ❷. If the credit counters of both

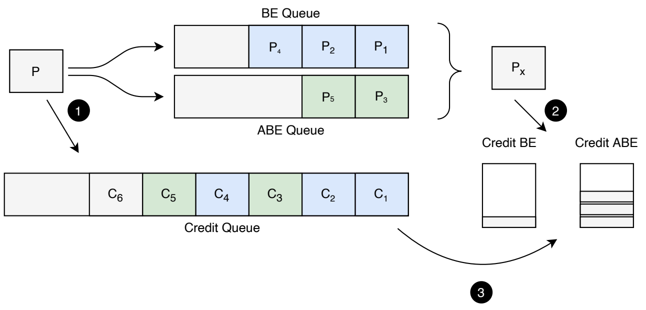


Fig. 1. The DSCD scheduler stores BE and ABE packets in the class-specific packet queues $Q[BE]$ and $Q[ABE]$. For every enqueued packet, a credit element is inserted into the credit queue Qc. The credit counters $CC[BE]$ and $CC[ABE]$ store class-specific credits that are needed for packet dequeue. If these credits do not suffice, new credits are taken from the credit queue.

queues are too low for dequeuing a packet, a credit element is removed from the credit queue and the credit counter of the element's class is incremented by the element's size ❸. The credit of dropped ABE packets remains in the system for some time. With that saved credit, subsequent ABE packets can be served earlier than comparable BE packets, but without delaying BE packets longer than in a pure FIFO system.

This sketch misses some details. First, packets may be lost during enqueue due to limited queue size. Second, low rates of ABE traffic should not experience high packet loss, which requires some extra logic. Third, credit should not be stored for infinite time, even in the presence of congestion. To that end, the credit is devaluated over time according to an exponential function ($exp(-\lambda \cdot \Delta)$) where Δ is the passed time and λ is the decay rate. We configure it via the half-life time $t_h = \frac{\ln(2)}{\lambda}$. After one half-life time t_h only half the credit is still available. Fourth, to simulate the behaviour of a FIFO queue, the credit should vanish when both packet queues are empty. It is drained from the system with link bandwidth C, which is continuously estimated.

B. Algorithm

We formalize the above sketched algorithm using pseudocode and address the missing details. We introduce the data structure of DSCD and describe its algorithms for packet enqueue, packet dequeue, credit devaluation, and bandwidth estimation.

1) Data structures: The data structures of DSCD are illustrated in Figure 1.

- DSCD maintains separate FIFO queues $Q[BE]$ and $Q[ABE]$ to store BE and ABE packets
- and a FIFO queue Qc to store credit elements.
- There are global counters for dequeued credit elements for each traffic class, $CC[BE]$ and $CC[ABE]$, which are zero at system start.
- The credit counter CC_{cq} counts the stored credit in the credit queue Qc.
- Furthermore, the global variable C stores the available bandwidth which is estimated in Algorithm 4 and utilized in Algorithm 3.

- Algorithm 3 also uses the global variable $lastDevaluation$ to record the last devaluation instant which is initialized with $-\infty$.
- Algorithm 4 uses global variables as helpers for bandwidth estimation. S_B, S_T, and $lastPktSize$ are initialized with zero, $backlogged$ with false, and $lastDequeue$ and $lastRateUpdate$ with $-\infty$.

2) DSCD Enqueue: The enqueue operation is given by Algorithm 1. First, saved credit is devaluated, which is described in Section III-B4. Then, a new packet p is dropped if its size together with the overall credit in the system exceeds the buffer size B_{max} (line 2-3). Otherwise, the packet is enqueued. Then, the deadline $p.d$ is set for ABE packets (line 5-6). In the remainder, an element e with the packet's length $p.len$ and class $p.class$ is created. It is added to the credit queue Qc whose credit counter CC_{cq} is incremented. Finally, the packet is added to its class-specific queue.

Algorithm 1: DSCD enqueue routine

```
Input  : Packet p
1  $DevaluateCredit()$
2  if $p.len + CC_{cq} + CC[BE] + CC[ABE] > B_{max}$ then
3      $drop(p)$
4  else
5      if $p.class == ABE$ then
6          $p.d = t_{now} + T_d$
7      e = new CreditElement(p.len, p.class)
8      Qc.add(e)
9      $CC_{cq}$ += p.len
10     Q[p.class].add(p)
```

3) DSCD Dequeue: We first explain the principle of the dequeue operation before we go into details.

DSCD has class-specific credit counters $CC[X]$, $X \in \{BE, ABE\}$. When a packet is dequeued, the corresponding credit counter is decreased by the packet size, but it cannot fall below zero. If both credit counters are too low to dequeue a packet, credit elements are removed from the credit queue and the elements' credit it added to the corresponding credit counters. A packet is dequeued as soon as one credit counter is large enough. If both counters are large enough, ABE traffic is preferentially served.

We now look at the pseudocode in Algorithm 2 which dequeues a packet if possible and returns NULL otherwise. First, the credit counter $CC[ABE]$ is devaluated, which is described in Algorithm 3. Afterwards, all ABE packets with violated deadlines are dropped from the ABE queue if they are followed by more than T_q other packets (line 2-4). The second part of the condition avoids packet loss when too few other packets in the packet queue could use the credit of dropped packets. Then, the return packet is initialized with NULL and a dequeue attempt is made only if the system holds at least one packet (line 6). The packet for dequeue is determined in the subsequent loop which ends with a successfully dequeued packet (line 7-16). Within the loop, a

Algorithm 2: DSCD dequeue routine

```
Output: Next packet to be served
1  $DevaluateCredit()$
2  while $Q[ABE].head.d > t_{now}$ and $Q[ABE].len > T_q$ do
3      $drop(Q[ABE].removeHead())$
4  end
5  $p = NULL$
6  if $!Q[BE].empty()$ or $!Q[ABE].empty()$ then
7      while $p == NULL$ do
8          if $CC[ABE] \geq Q[ABE].head.len$ then
9              $p = Q[ABE].removeHead()$
10         else if $CC[BE] \geq Q[BE].head.len$ then
11             $p = Q[BE].removeHead()$
12         else
13             $e = Qc.removeHead()$
14             $CC_{cq} -= e.credit$
15             $CC[e.class] += e.credit$
16     end
17     $CC[p.class] -= p.len$
18     $EstimateBandwidth(p)$
19 return $p$
```

queue with a sufficiently large credit counter is determined and its first packet is removed (line 8-11). If neither queue has a sufficiently large credit counter, a credit element is removed from the credit queue and the credit counter of the corresponding traffic class is incremented (line 13-15). After successful packet dequeue, the credit counter of the respective class is decremented (line 17). Then, the estimate of the link bandwidth C is updated using Algorithm 4 (line 18). Finally, either the dequeued packet or a NULL pointer is returned.

4) Credit Devaluation: Credit devaluation is needed for two reasons.

First, the overall credit in the system, i.e., the credit in the credit queue Q_c and the counters $CC[BE]$ and $CC[ABE]$, simulates an upper bound of the fill state of an alternative FIFO queue. That is a necessary invariant to ensure that BE packets are not served later than in a comparable FIFO queue. If both packet queues are empty, remaining credit in the system must vanish with the current link bandwidth C.

Second, credit from dropped ABE packets is stored by $CC[ABE]$ and used to send other ABE packets early. Without additional devaluation, credit from dropped ABE packets remains in the system until the end of a congestion period. This may incentivize applications to send unnecessary ABE data to provoke packet loss and leverage resulting credit in order to gain a delay advantage for later ABE traffic. Therefore, we believe that credit should vanish over time, even in the presence of congestion. Further, transport protocols benefiting from lower transmission delay may obtain a throughput advantage via ABE compared to those transmitting over BE, despite increased packet loss. Credit devaluation limits that advantage (see Section V-E) and, thereby, leads to better inter-class fairness for transport protocols.

As pointed out, credit devaluation is needed in two ways: (1) If the system is empty, ABE's saved credit $CC[ABE]$ is reduced over time by the transmission rate C. (2) ABE's saved credit $CC[ABE]$ decays over time exponentially with rate λ. This pursues the previously discussed objectives.

Algorithm 3: DevaluateCredit

```
1  $\Delta = t_{now} - lastDevaluation$
2  $lastDevaluation = t_{now}$
3  **if** $Q[BE].empty()$ *and* $Q[ABE].empty()$ **then**
4      **while** $!Qc.empty()$ **do**
5          $e = Qc.removeHead()$
6          $CC_{cq} -= e.credit$
7          $CC[e.class] += e.credit$
8      $CC[ABE] = max(0, CC[ABE] - C \cdot \Delta)$
9  **else**
10     $CC[ABE] = CC[ABE] \cdot exp(-\lambda \cdot \Delta)$
```

Algorithm 3 performs these operations. First, the passed time Δ since the last devaluation is computed and the global variable $lastDevaluation$ is updated by the current time t_{now}. If both queues are empty, the credit queue is emptied and the credit counters of the corresponding packet queues are incremented (line 4-7). Then, ABE's credit counter $CC[ABE]$ is reduced with the current bandwidth C over time Δ; thereby the credit counter cannot fall below zero. If at least one queue is not empty, ABE's credit counter $CC[ABE]$ is devaluated exponentially over time Δ with rate λ (line 10). We call this mechanism *exponential decay*.

5) Bandwidth Estimation: Algorithm 3 requires an estimate C of the link bandwidth to devaluate credit in the presence of an empty queue. To measure C, an amount of sent bytes is divided by their transmission time. We capture the transmission time of a packet from the time it is dequeued until the next packet is dequeued, provided that there is no idle time in between. We ensure this by considering only packets that leave a non-empty queue, i.e., a backlogged queue.

To cope with varying bandwidth, we accumulate both sent bytes and transmission times by weighted sums S_B and S_T and derive an estimate by $C = \frac{S_B}{S_T}$. We utilize the weighted sum of the moving average UTEMA [21] for that purpose:

$$S_X(t) = S(t_{last}) \cdot e^{-\mu \cdot (t - t_{last})} + X \quad (1)$$

$$t_{last} = t. \quad (2)$$

X is a series of samples at time instants t. $S_X(t)$ is the weighted sum of the sampels at time t. The sum is updated whenever a new sample is available and t_{last} indicates the last update time of the sum. The advantage of UTEMA is that the contribution of the samples considered in the weighted sum decreases exponentially over time with rate μ, i.e., newer samples have a larger impact on the sum than older samples. We apply this concept to the size of sent packets B and their mere transmission times T, which yields S_B and S_T. For configuration, a memory M is used to set the rate $\mu = \frac{1}{M}$.

Algorithm 4: EstimateBandwidth

```
   **Input** : *Packet p*
1  **if** $backlogged$ **then**
2      $\Delta = t_{now} - lastRateUpdate$
3      $S_B = S_B \cdot exp(-\mu \cdot \Delta) + lastPktSize$
4      $S_T = S_T \cdot exp(-\mu \cdot \Delta) + (t_{now} - lastDequeue)$
5      $C = S_B / S_T$
6      $lastRateUpdate = t_{now}$
7  **if** $Q[ABE].len + Q[BE].len > 0$ **then**
8      $backlogged = true$
9  **else**
10     $backlogged = false$
11 $lastDequeue = t_{now}$
12 $lastPktSize = p.len$
```

Algorithm 4 translates this concept into pseudocode for a rate estimation procedure that is called at the end of each successful packet dequeue. In the first step of the algorithm (line 1-6), the estimated rate C is updated if the last dequeued packet was backlogged. The elapsed time Δ since the last rate update is computed and used to devaluate the weighted sums of bytes and transmission times (S_B, S_T) which are also increased by the size of the last dequeued packet ($lastPktSize$) and its transmission time ($t_{now} - lastDequeue$). Then, the estimated bandwidth C and the last rate update time $lastRateUpdate$ are updated. The variable $backlogged$ is set to true if there are more packets waiting in some queue, otherwise it is set to false (line 7-10). Finally, the current dequeue time and the size of the dequeued packet are recorded by $lastDequeue$ and $lastPktSize$.

IV. Implementation of DSCD in the Linux Network Stack

Traffic schedulers and AQMs are often evaluated using simulation frameworks such as ns-3 or OMNeT++.

Although simulation frameworks offer a lot of freedom regarding implementation, they are only an approximation of reality. Therefore, the trustworthiness of simulation results for complex protocols, such as TCP, heavily depend on the validity of the simulation model. For this reason, we decided to implement DSCD in the Linux network stack as proof-of-concept implementation and perform experiments with existing protocol implementations, in particular up-to-date TCP variants. Moreover, this implementation demonstrates the practical feasibility of DSCD.

We first introduce some background information on queuing disciplines in the Linux kernel. Afterwards, we present an efficient approximation of an exponential decay function for the Linux kernel. Then we explain the implementation of the exponential decay for the stored ABE credit and the rate estimation as these implementation aspects are most challenging. The overall code for DSCD on Linux is available at Github[1].

[1]https://github.com/uni-tue-kn/dscd-linux-qdisc

A. Use of Queuing Disciplines in the Linux Kernel

Queuing disciplines, also called QDiscs, are part of the Linux network stack and are located in the kernel space. They perform tasks such as traffic shaping, packet classification, or packet dropping. A self-implemented QDisc may perform other, almost arbitrary operations on packets. QDiscs are implemented in the C programming language.

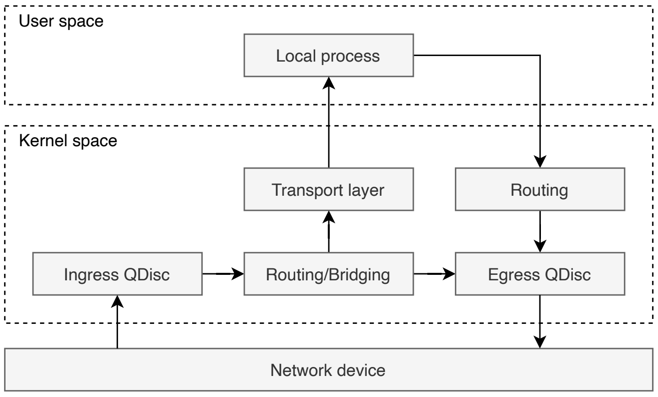


Fig. 2. Packet handling in the Linux network stack.

Figure 2 illustrates the simplified packet handling in the Linux network stack. Incoming packets from the network card are passed to the Linux kernel. Initially, packets are handed to an ingress QDisc. Within an ingress QDisc, packets can be filtered or rate-limited. Afterwards, a routing or bridging decision is taken. If the packet is destined for the host itself, the packet is passed to the transport layer and further to the application process in the user space. Otherwise, the packet is passed to the egress QDisc of the outgoing interface. DSCD is completely implemented as egress QDisc. Both ingress and egress QDiscs provide a standardized interface to the Linux kernel. It includes functions for packet enqueue and dequeue which correspond to the algorithms presented in Section III.

QDiscs also provide the possibility for *chaining*. This means that multiple QDiscs are executed one after another. Chaining is used to separate functionality between different QDiscs, e.g., rate-limiting and classification. QDiscs leveraging chaining are called *classful* and are organized in a tree structure. The Kernel enqueues the packet in the so-called root QDisc. The root QDisc then enqueues the packet into one of its child QDiscs which may enqueue the packet in one of its own child QDiscs. When a packet should be dequeued, the Kernel calls the dequeue routine at the root QDisc which in turn calls the dequeue routine of its child QDiscs.

We leverage the functionality of classful QDiscs within our testbed to combine rate-limiting (with the classful QDisc tbf) and our DSCD QDisc. The use of tbf facilitates the configuration of a controlled and variable bottleneck capacity as described in Section V-A1.

B. Efficient Approximation of the Exponential Function

The algorithms presented in Section III require the computation of an exponential function for credit devaluation and rate estimation. However, the exponential function is not available in the kernel and only integer arithmetic can be used[2]. Therefore, we present an approximation for the multiplication of an integer n with $exp(x)$ that can be efficiently implemented in the Linux kernel. The exponential function can be rewritten as

$$exp(-x) = 2^{-x/ln(2)}. \tag{3}$$

We first propose a piecewise linear function as floating point approximation of 2^{-x} and then we implement $n \cdot 2^{-x}$ with integer arithmetic. Finally, we consider the application of the approximation to exponential decay and bandwidth estimation.

1) Floating Point Approximation of 2^{-x} for Positive Arguments: To approximate the power function $p(x) = 2^{-x}$, we use the following piecewise linear function for $x \geq 0$ which uses interpolation of integer-based sampling points only:

$$f(x) = \frac{\lfloor x \rfloor - x + 2}{2^{\lfloor x \rfloor + 1}}. \tag{4}$$

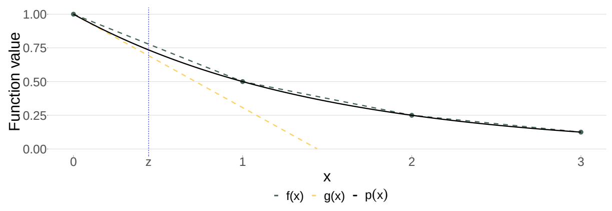
We improve the error for small values of x by another approximation

$$g(x) = 1 - ln(2) \cdot x \tag{5}$$

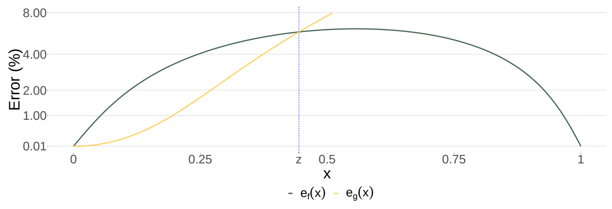
which is based on the derivative of 2^{-x} at $x = 0$. Both approximations are illustrated in Figure 3(a) and the corresponding error functions in Figure 3(b). We combine them to minimize the error by

$$h(x) = \begin{cases} g(x) & 0 \leq x \leq z \\ f(x) & z < x \end{cases} \tag{6}$$

with $z \approx 0.4443$ being the abscissa of the intersection point of both error functions.



(a) Approximation of the power function $p(x) = 2^{-x}$ by the piecewise linear function $f(x)$ and the derivative-based linear function $g(x)$.



(b) Error functions $e_f(x) = f(x) - 2^{-x}$ for the piecewise linear approximation and $e_g = g(x) - 2^{-x}$ for the derivative-based linear approximation.

Fig. 3. Approximation options for the power function $p(x) = 2^{-x}$. We combine them in $h(x)$ to minimize the error.

[2]https://www.kernel.org/doc/html/v5.0/process/howto.html

2) Implementation of $n \cdot 2^{-x}$ with Integer Arithmetic: The argument for the power function is a fractional number x. As the Linux kernel supports only integer arithmetic, we represent the argument by a scaled number $y = x \cdot 2^s$; we denote s the scaling exponent.

We propose the function $a(n, y, s) = n \cdot 2^{(-y/2^s)}$ to multiply an integer n with a power function value where n, y, and s are non-negative integers. We implement $a(n, y, s)$ using function h with argument $y/2^s$ and utilize the approximations

$$ln(2) \approx \frac{2^{12}}{5909} \quad (7)$$

$$z \approx \frac{2^{12}}{9219}. \quad (8)$$

This results in

$$a(n,y,s) = \begin{cases} n - \dfrac{y \cdot n}{5909 \cdot 2^{s-12}} & y \cdot 9219 \leq 2^{s+12} \\ \dfrac{n \cdot (\frac{y}{2^s} + 2) - \frac{n \cdot y}{2^s}}{2^{(\frac{y}{2^s}+1)}} & 2^{s+12} < y \cdot 9219 \end{cases}. \quad (9)$$

Here, divisions imply integer divisions. Therefore, we can substitute $\lfloor x \rfloor$ in $h(x)$ by $\frac{y}{2^s}$ in the formula. We first evaluate numerator and denominator of any fraction prior to division to prevent unnecessary loss of accuracy. For efficiency, every division by 2^k is performed as a bit shift by k bits to the right. Moreover, intermediate results may be stored and reused.

The computation of $n \cdot 2^{-x}$ is achieved by calling $a(n, y, s)$ with the arguments n, $y = x \cdot 2^s$, and s. For the computation of $n \cdot exp(-x)$ Equations (3) and (7) need to be taken into account so that $a(n, y, s)$ is to be called with arguments n, $y = x \cdot 5909 \cdot 2^{s-12}$, and s.

If a floating point number m is to be multiplied with an exponential value, one can scale m to $n = m \cdot 2^{(s_m)}$ with a scaling exponent s_m before applying it to $a(n, y, s)$. The returned number is the result scaled with $2^{(s_m)}$.

3) Application to Exponential Decay and Bandwidth Estimation: We apply $a(n, y, s)$ to implement line 10 in Algorithm 3 (DevaluateCredit) and line 3-4 in Algorithm 4 (EstimateBandwidth). For DevaluateCredit, the rate $\lambda = \ln(2)/t_h$ is configured via the half-life time t_h. Both Δ and t_h are counted in ns. The parameter $y = \Delta/t_h$ is additionally scaled with 2^{20}, i.e., $s = 20$, to gain precision for small values of Δ. Credits are scaled with 2^{10} to limit loss of accuracy for small integers, i.e., $s_m = 10$. For EstimateBandwidth, the rate $\mu = 1/M$ is configured via the memory M. Both Δ and M are counted in ns. The parameter $y = \mu \cdot \Delta/\ln(2)$ is again scaled with 2^{20}, i.e., $s = 20$.

In the following, we derive an upper bound for the relative error in practise. The transmission time for packets with 1490 bytes is 1.2 ms with 10 Mbit/s and 12 µs for 1 Gbit/s. The algorithms are mostly called in these intervals. The half-life time for exponential decay of $t_h = 100$ ms and a memory for bandwidth estimation of $M = 50$ ms correspond to rates of $\lambda = \frac{ln(2)}{t_h} = \frac{6.9}{\text{s}}$ and $\mu = \frac{1}{M} = \frac{20}{\text{s}}$. Therefore, the exponential function is called under relevant conditions with values smaller than $1.2 \text{ ms} \cdot \frac{20}{\text{s}} = 0.024$ and the approximation $h(x)$ is called with values smaller than $\frac{0.024}{ln(2)} = 0.034$. The relative error by the approximation for such values is about 0.029% (see Figure 3(b)), i.e., very low. For higher link speeds, e.g., 10 Gbit/s or 100 Gbit/s, the relative error further decreases.

C. Performance of Linux QDisc Forwarding

Modern NICs support multiple transmit (TX) and receive (RX) queues to facilitate highspeed packet processing. Packets are distributed to different queues such that they can be processed by separate CPU cores without interference, e.g., caused by lock mechanisms. This mechanism is called Receive-Side Scaling (RSS). Packets are assigned to a queue using a hash function, e.g., 4-tuple hash over IP addresses and TCP ports[3]. In the following, we investigate the general performance for traffic forwarding with Linux QDiscs. We deploy the pfifo QDisc in a similar testbed as presented in Section V-A1. The bottleneck link has a capacity of 100 Gbit/s. We vary the number of available CPU cores and TX queues[4] and establish 32 TCP flows between the senders and the receiver. Table II shows the L2 throughput, i.e., Ethernet throughput, on the bottleneck.

TABLE II

#CPU cores	#TX queues	L2 throughput (Gbit/s)
1	1	25.86
2	1	34.49
	2	48.45
4	1	38.05
	2	62.73
	4	92.84
8	1	40.29
	2	68.42
	4	96.72
	8	97.07

With a single CPU core (and a single TX queue), only 25.86 Gbit/s can be achieved on L2. The throughput with a single TX queue can be increased by increasing the number of CPU cores. However, the throughput does not linearly increase with the number of CPU cores and converges at 40.29 Gbit/s for 8 CPU cores. This is caused by communication overhead between the CPU cores as the TX queue can only be accessed by a single CPU core at a time. The throughput increases with an increasing number of TX queues and converges at 97.07 Gbit/s for 8 CPU cores with 8 TX queues. A saturation below 100 Gbit/s is reasonable as the L2 throughput does not include preamble and inter-frame gap. The experiment shows that forwarding with 100 Gbit/s requires multiple CPU cores and TX queues on Linux systems. Therefore, we use 8 CPU cores and 8 TX queues when performing experiments at 100 Gbit/s.

As the assignment of packets to TX queues is based on a hash function, packets of the same flow are placed in the same TX queue. If only a subset of the TX queues are used by chance, 100 Gb/s may not be achieved. However, this is unlikely in a 100 Gb/s environment where the number of flows is high.

[3]https://www.kernel.org/doc/Documentation/networking/scaling.txt

[4]The maximum number of TX queues is limited by the number of available CPU cores.

D. *Efficiency of the DSCD Implementation*

Now, we assess the efficiency of the DSCD implementation. At first sight, the DSCD algorithm has some complexity, but the implementation is efficient. We demonstrate that by the following experiments.

We deploy DSCD in a similar testbed[5] as presented in Section V-A1. The bottleneck link has a capacity of 100 Gbit/s. The delay threshold is set to $T_d = 10$ ms and the half-life time t_h is set to $t_h = 100$ ms. We establish 32 TCP flows between them. Every 2nd TCP flow is labeled as ABE. We measure the overall TCP goodput and average CPU load of DSCD and compare it to existing Linux QDiscs[6] such as FQ-CoDel, FQ-PIE, Stochastic Fair Queuing (SFQ), and pfifo. Table III shows the results.

TABLE III
TCP GOODPUT AND CPU LOAD OF VARIOUS LINUX QDISCS.

QDisc	TCP goodput (Gbit/s)	CPU load (%)
DSCD	89.08	36.27
FQ-CoDel	89.02	38.99
FQ-PIE	89.00	44.21
SFQ	89.03	38.72
pfifo	89.06	35.41

All considered QDiscs achieve approximately the same goodput of ~ 89 Gbit/s. pfifo and DSCD have the lowest CPU load with 35.41% and 36.27% and FQ-PIE the highest with 44.21%. The results show that the DSCD implementation in the Linux kernel is efficient and comparable to other QDiscs.

V. PERFORMANCE EVALUATION

We first explain our performance evaluation methodology. Then we validate the bandwidth estimation algorithm which is part of DSCD. Afterwards, we study DSCD scheduling for non-adaptive traffic, periodic traffic, and TCP traffic, demonstrating the impact of configuration parameters and ABE traffic rates on packet loss and delay. Finally, we investigate inter-protocol and inter-class fairness for different TCP variants in connection with ABE.

A. *Methodology*

We introduce the methodology for the performance study. We present the testbed, explain experiment organization and performance metrics, and describe how traffic is generated.

1) Testbed: We leverage a semi-virtualized testbed on a host system with 128 GB RAM. We work with KVM-based virtual machines (VMs) that are assigned 4 GB RAM and two cores with 3.2 GHz from an Intel(R) Xeon(R) Gold 6134. The VMs run with Linux kernel 5.10 and have dedicated 10 Gbit/s network cards. Thus, if not mentioned differently, links between VMs have a capacity of exactly 10 Gbit/s. No overbooking is performed on the VM hosts, the NICs are passed through to the VMs using SR-IOV and the CPUs are pinned to physical cores to minimize the influence of virtualization on the experimental results.

[5]The bottleneck VM has 8 cores in this experiment.

[6]We use multi-queuing (8 TX queues) to improve performance.

The logical structure of the testbed is illustrated in Figure 4. Up to five VMs send traffic to a bottleneck VM via dedicated 10 Gbit/s links. The bottleneck VM is connected to a so-called RTT VM via a throttled link which has a capacity of $C = 1$ Gbit/s in most experiments. This is done to perform experiments with software generated traffic (see Section V-A3) and to study DSCDs behavior in a controlled environment. However, as shown in Section IV-D, DSCD supports much higher bandwidths. The rate limitation is achieved with the Linux queuing discipline *tbf* [22] using a rate of 1 Gbit/s and an tbf bucket size of 10 maximum transfer units (MTUs)[7]. This represents the bottleneck of the path and possibly causes congestion. DSCD is deployed at the bottleneck node with a buffer size of $B_{max} = C \cdot 25$ ms $= 3.125$ MB. Thus, packets are queued by DSCD and sent whenever tbf allows. Further default DSCD parameters are a delay threshold of $T_d = 10$ ms, a half-life time of $t_h = 100$ ms, and a queue threshold of $T_q = 1$.

The RTT VM delays packets according to a configured RTT. The Linux queuing discipline Netem [23] is utilized to delay the traffic by RTT time for which the default value is $RTT = 100$ ms.

Table IV summarizes the default configuration for the experiments. Rate limitation by tbf, DSCD scheduling, and delay addition are applied only in one direction.

TABLE IV
DEFAULT CONFIGURATION OF TESTBED AND DSCD ALGORITHM.

Parameter	C	RTT	B_{max}	T_d	t_h	T_q
Value	1 Gbit/s	100 ms	25 ms	10 ms	100 ms	1

2) Performance Metrics and Experiment Organization: The performance metrics in the experiments are packet queuing delay and packet loss on the bottleneck node and end-to-end goodput of TCP flows.

Every experiment, i.e., a set of studied parameters, is executed 30 times and runs for 45 s. Data from the first 15 s of each run are discarded to avoid the impact of a potential transient phase. Data from the last 2 s are removed as not all streams may be terminated simultaneously. For the remaining 28 s we calculate performance metrics. We average them over the 30 runs and calculate 95% confidence intervals. However, we omit them in the figures for the sake of readability as they are very small.

3) Traffic Generation: In the experiments, three different traffic types are utilized. We describe their generation in the following.

a) Non-adaptive traffic with bursts: To investigate basic effects of DSCD scheduling without interactions of transport protocols such as TCP, we apply non-adaptive traffic with bursts. Poisson traffic is a first candidate but does not cause substantial queuing at link speeds of 1 Gbit/s. Therefore, we generate packets with LogNormal-distributed packet inter-arrival times and send them over UDP.

[7]We set the bucket size to a low value to avoid substantial impact on DSCD queuing. We validated that a bandwidth of 1 Gb/s is still achieved with this setting.

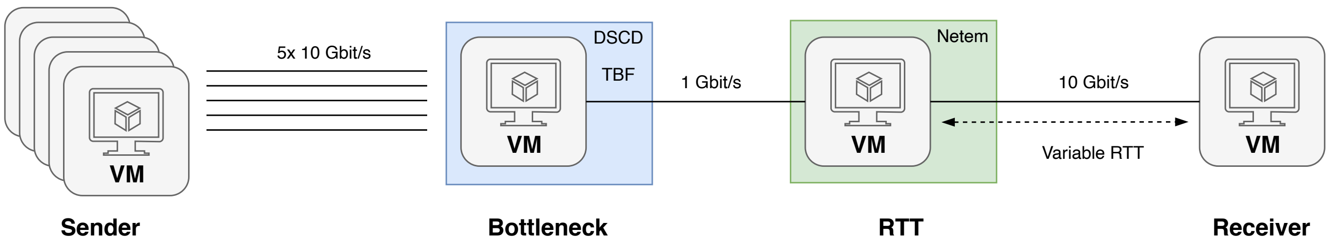


Fig. 4. The performance evaluation is carried out in a semi-virtualiezd testbed. It consists of multiple virtual machines (VMs) on a single server. The VMs are assigned dedicated 10 Gbit/s network cards. Up to five VMs send traffic to a receiver. The path has a bottleneck of 1 Gbit/s and a configurable RTT that are imposed by a bottleneck node and an RTT node. DSCD is deployed at the bottleneck node.

We derive mean and standard deviation of the random variable A, which denotes the inter-arrival time, from the following desired properties. Let C be the capacity of the link and ρ its target utilization. Given a fixed packet size of $B = 1490$ bytes including all headers (20 bytes for IP, 8 bytes for UDP, 14 bytes for Ethernet), we can compute the number of packets N within a 10 ms interval by $N = \rho \cdot C \cdot 10\ \text{ms}/B$. Thus, the expected packet inter-arrival time is $E[A] = 10\ \ \text{ms}/N$. We set the standard deviation $\sigma[A]$ such that the standard deviation of N inter-arrival times is 5 ms. Thus, the standard deviation of a single inter-arrival time is $\sigma[A] = 5\ \text{ms}/\sqrt{N}$. This traffic is sufficiently bursty. We generate it on a single machine using 40 threads that send packets in a round-robin manner.

To visualize the effect of the resulting arrival process, we experimentally count the number of arrived packets within a 10 ms interval. Figure 5 shows the cumulative distribution function (CDF) of that number for a relative load $\rho \in \{0.95, 1.2\}$ on a link with a capacity of 1 Gb/s. We observe that the number of packets arrived within a 10 ms interval vary substantially around their means (dashed lines). Moreover, there are many 10 ms intervals with clearly less and more traffic arrived than what could be sent within that time (solid line). As a consequence, the generated traffic is bursty and leads to substantial packet queuing. The offered load $\rho = 0.95$ models moderate overload and $\rho = 1.2$ models severe overload. The discussed arrival processes are utilized in the experiments of Section V-C1 and Section V-C2.

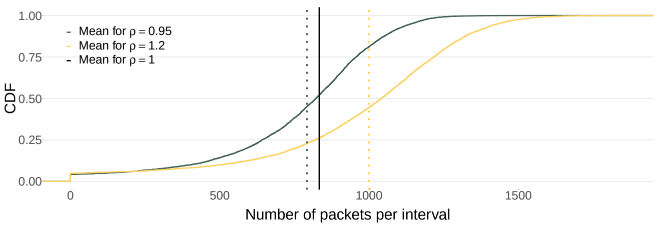


Fig. 5. Cumulative distribution function (CDF) of the number of packets arrived within a 10 ms interval; the LogNormal-distributed inter-arrival times A are set for a relative load of $\rho \in \{0.95, 1.2\}$ on a link with a capacity of $C = 1$ Gbit/s; the vertical lines correspond to mean rates and the link bandwidth.

b) Constant bit-rate traffic UDP traffic: Realtime traffic sent over UDP can be often modelled as periodic constant bit-rate (CBR) traffic. It is a typical candidate to benefit from the ABE traffic class. We leverage *iperf3.9* [24] for the generation of CBR traffic and send it over UDP. The packet size is $B = 1490$ bytes including all headers, and constant inter-arrival times are set to achieve a desired traffic rate.

c) Elastic TCP traffic: Most traffic on the Internet is transmitted over TCP which adapts its transmission rate to the congestion conditions in the network in order to avoid excessive packet loss. It is also called elastic traffic as it utilizes the available bandwidth. Various TCP versions exist and have different properties. Some react primarily to packet loss, e.g., TCP Cubic, others react to increased RTT, e.g., TCP BBR. We leverage *iperf3.9* [24] for the generation of TCP traffic. We utilize both TCP Cubic and TCP BBR by choosing appropriate Linux implementations in the VMs.

B. Validation of the Bandwidth Estimation Algorithm

In Section III-B5 we presented a new bandwidth estimation algorithm. For this study, we set its memory to $M = 50$ ms. We validate the method with the following experiment. We send non-adaptive traffic with bursts as described in Section V-A3 with an load of $\rho = 0.5$ on a link with 1 Gbit/s. The link bandwidth is set by tbf using a burst size of 10 MTUs like in all other experiments. After 5 seconds, the bottleneck decreases to 250 Mbit/s and changes back to 1 Gbit/s after 7 seconds.

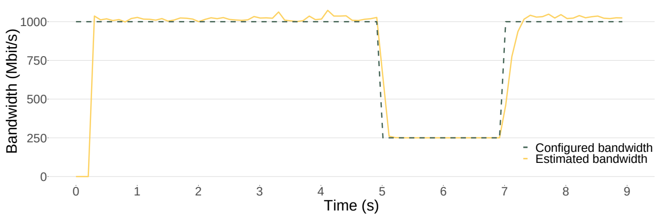
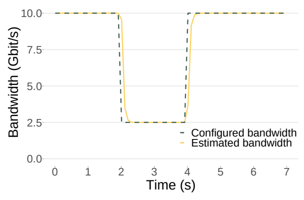


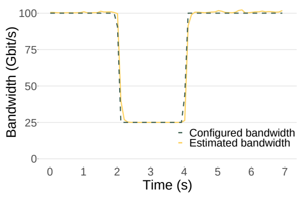
Fig. 6. Bandwidth estimation on a link where tbf-controlled bandwidth starts with 1 Gbit/s, it is 250 Mbit/s after 5 s, and changes back to 1 Gbit/s after 7 s. Non-adaptive traffic with bursts is sent at a rate of 500 Mbit/s.

Figure 6 shows that the estimated bottleneck matches the configured bandwidth very closely. The challenge is the adaptation at 7 s to a larger rate as the utilization is then only 50%. Then, backlogged packets do not occur often, but they are frequent enough as the algorithm leverages every single backlogged packet to update its estimate. Thus, the estimation method is very sensitive in the sense that it recognizes the correct rate even under moderate load. The algorithm works equally well with TCP traffic (32 flows) which is shown in

Figure 7(a) and Figure 7(b) for a link with 10 Gbit/s and 100 Gbit/s capacity. Again, the bottleneck changes to 25% of its original capacity after 2 seconds, and changes back after 4 seconds.



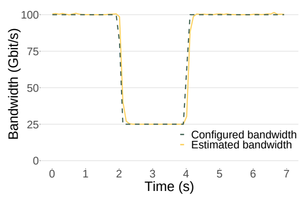
(a) Link bandwidth 10 Gbit/s.



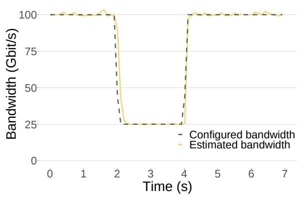
(b) Link bandwidth 100 Gbit/s.

Fig. 7. Bandwidth estimation with 32 TCP flows; the link bandwidth is throttled to 2.5 (25) Gbit/s between 2 s and 4 s.

The bandwidth estimation algorithm works equally well with 8 and 16 TCP flows[8] as shown in Figure 8(a) and Figure 8(b). With a lower number than 8 TCP flows, there is



(a) 8 TCP flows.



(b) 16 TCP flows.

Fig. 8. Bandwidth estimation with 8 and 16 TCP flows; the link bandwidth is throttled to 25 Gbit/s between 2 s and 4 s.

not sufficient congestion to trigger the bandwidth estimation algorithm. However, this is not a problem for the following reasons. First, less than 8 TCP flows in a 100 Gbit/s environment is rather unlikely. Second, in the absence of congestion, DSCD does not require the estimated bandwidth C. The estimated bandwidth is only used for credit devaluation after a congestion period.

The experiments show that the bandwidth estimation algorithm precisely measures the available bandwidth with the same parameterization (M = 50 ms) for a wide range of bottleneck speeds.

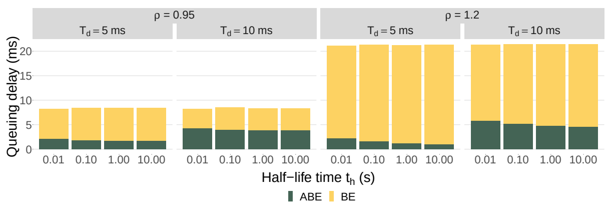
C. Performance of DSCD with Non-Adaptive Traffic with Bursts

We study packet delay and loss for non-adaptive traffic with bursts when being carried over BE and ABE. We first study the impact of DSCD parameters T_d, t_h and then the impact of ABE traffic rate at different link loads ρ.

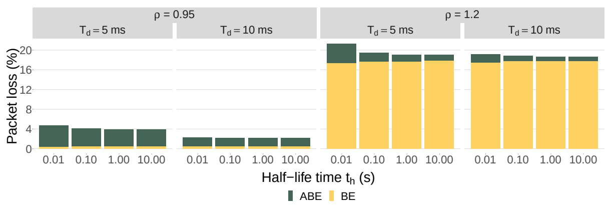
1) Impact of DSCD's Configuration Parameters T_d and t_h: DSCD is configured with two parameters: the delay threshold T_d for ABE traffic and the half-life time t_h for credit devaluation. We examine their impact with the following experiment. We generate non-adaptive traffic with bursts as explained in Section V-A3 with an offered load of $\rho \in \{0.95, 1.2\}$. We randomly label 90% of the traffic as BE and 10% as ABE.

[8]As long as all TX queues are used.

We experiment with different delay thresholds T_d and half-life times t_h. Other parameters are set to the default values in Table IV. We study the experienced queuing delay and packet loss at the bottleneck node separately for ABE and BE traffic. Figure 9(a) and Figure 9(b) illustrate the results.



(a) Queuing delay.



(b) Packet loss.

Fig. 9. Queuing delay and packet loss of BE and ABE traffic for non-adaptive traffic with bursts. The relative overall load ρ, the delay threshold T_d, and the half-life time t_h are varying parameters; other parameters are set as in Table IV.

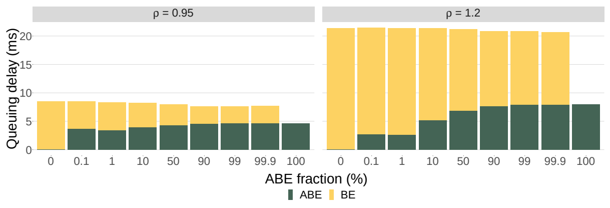
We first analyze the delay. Figure 9(a) shows that both BE and ABE traffic is only little delayed with moderate overload ($\rho = 0.95$), but ABE traffic experiences less delay than BE traffic. A lower delay threshold T_d reduces the delay for ABE traffic. For severe overload ($\rho = 1.2$), BE traffic is strongly delayed while ABE traffic sees similarly low delays as for moderate overload. Larger half-life times slightly reduce the delay for ABE traffic.

Now, we discuss the packet loss. Figure 9(b) shows that in the presence of moderate overload hardly any BE packets are lost while the packet loss probability for ABE traffic is 2%–4%. The lower the delay threshold T_d, the higher the packet loss. The additional packet loss of ABE is caused by the traffic model. With an offered load of $\rho = 0.95$ the traffic model results in either no congestion (empty queue) or congestion induced by bursts. In the case of no congestion, the stored ABE credit is devaluated with the link rate R. As a result, with $\rho = 0.95$ it is likely that ABE traffic is not able to "save" credit between bursts, i.e., each burst arrives at an empty ABE credit counter.

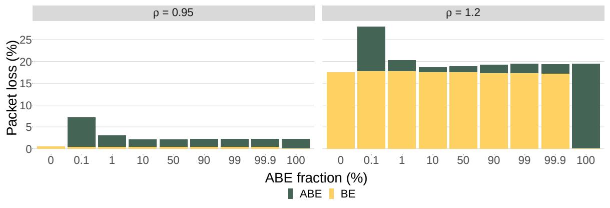
In case of severe overload, $\frac{1}{6}$ of the traffic cannot be carried due to missing capacity. Therefore, both BE and ABE traffic experience around 17% packet loss at packet enqueue. ABE traffic faces 1%–4% more packet loss than BE traffic, which is mostly due to exceeded deadlines. Larger half-life times slightly reduce the packet loss for ABE traffic. A half-life time of $t_h = 100$ ms leads to clearly less packet loss than $t_h = 10$ ms for severe overload. Longer half-life times lead only to minor improvements. Therefore, we recommend to set the half-life time to $t_h = 100$ ms. This will be confirmed in Section V-E for other reasons.

2) Impact of ABE Traffic Rate: DSCD turns dropped ABE packets into a potential delay advantage for subsequent ABE packets. If no such packets arrive in time, packet drops cannot be leveraged by the ABE traffic class. This may happen in the presence of too little ABE traffic. Therefore, we study the impact of ABE traffic rate on loss and delay.

The experiments are designed similarly as those in Section V-C1. Non-adaptive traffic with bursts is used with the standard configuration of Table IV. We study again an offered load of $\rho \in \{0.95, 1.2\}$ and test different ABE traffic rates by varying the fraction of ABE traffic.



(a) Queuing delay



(b) Packet loss

Fig. 10. Queuing delay and packet loss of BE and ABE traffic for non-adaptive traffic with bursts. The relative overall load ρ and the fraction of ABE traffic are varying parameters; other parameters are set as in Table IV.

We first discuss the packet loss in Figure 10(b). In the absence of ABE traffic, there is no ABE packet loss and only the little BE packet loss is visible. A very small fraction of ABE traffic ($0.1\% \approx 1$ Mbit/s) results in high packet loss of almost 7% and 27% for moderate and severe overload. When the ABE traffic rate is low, packet inter-arrival times are large. When an ABE packet exceeds the delay threshold T_d and the queue threshold T_q, it is dropped but its credit is saved. However, the credit is devaluated over time, either exponentially or linearly. Therefore, saved credit is likely to be vanished by the arrival of the next packet due to the low ABE traffic rate. Then, the next packet may also experience normal queueing delay, exceed the delay threshold, and be lost again.

For non-responsive bursty traffic, the packet loss is very high at a rate of 1 Mbit/s (ABE fraction $\approx 0.1\%$) in spite of the queue threshold $T_q = 1$. In Section V-D2 we will show that this setting is able to prevent excessive packet loss for periodic traffic of that rate. Luckily, realtime traffic is usually periodic. Further, ABE packet loss decreases with a larger fraction of ABE traffic as subsequent packets arrive earlier and can leverage stored credit more efficiently. An ABE traffic rate of 10 Mbit/s (ABE fraction $\approx 1\%$) suffices to achieve significantly lower packet loss.

The packet loss for BE traffic slightly decreases with a larger ABE fraction. With a larger ABE fraction, more ABE traffic is dropped, which reduces load from the system. BE traffic benefits from that with slightly reduced packet loss. In the absence of BE traffic, there is no BE packet loss and only the ABE packet loss is visible.

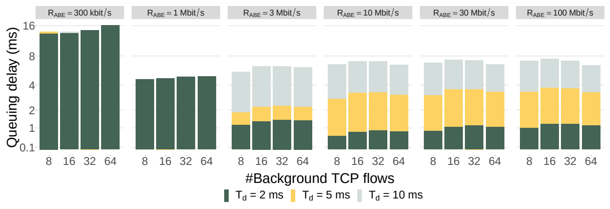
Figure 10(a) shows that queuing delay for ABE increases with larger ABE fraction. ABE packets can only overtake BE packets. Therefore, an increasing amount of ABE leads to more non-skippable packets and hence to longer queuing delay. Nevertheless, the delay is below the delay threshold T_d. At the same time, BE queuing delay decreases for increasing ABE fraction. This is due to reduced traffic load in the system as more traffic is dropped with larger ABE fraction.

The experiments show that even large fractions of ABE traffic have no negative impact on the performane of BE traffic, which was a design goal for ABE. This is unlike Expedited Forwarding (EF) of the differentiated services framework (DiffServ) [8] where BE traffic suffers if the fraction of EF traffic is too large.

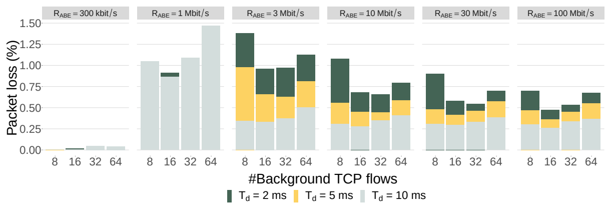
D. Performance of DSCD with Periodic Traffic and TCP Traffic

Now we assume that ABE traffic is periodic UDP traffic and BE traffic consists of TCP Cubic flows. This is a more realistic assumption as many realtime applications send periodic traffic. We first study packet delay and loss for different delay thresholds T_d, ABE traffic rates, and various numbers of TCP flows. Then we focus on small ABE traffic rates and show that the queue threshold $T_q = 1$ is the right means to prevent excessive packet loss.

1) Coexistence of Realtime and Elastic Traffic: We evaluate different sending rates R_{ABE} for ABE traffic and different numbers of TCP flows. We vary the delay threshold T_d and use the default settings from Table IV for other parameters.



(a) Queuing delay



(b) Packet loss

Fig. 11. Queuing delay and packet loss of periodic ABE traffic; ABE traffic rate R_{ABE}, number of TCP flows carried over BE, and delay threshold T_d are varying parameters; other parameters are set as in Table IV.

We first consider the ABE packet loss illustrated in Figure 11(b). For $R_{ABE} = 300$ kbit/s ABE packet loss is very low, for $R_{ABE} = 1$ Mbit/s it is large but almost independent of the delay threshold T_d, and for larger ABE traffic rates the packet loss depends on the delay threshold

T_d. This can be explained as follows. For a very low ABE traffic rate $R_{ABE} = 300$ kbit/s, the inter-arrival time of periodic ABE packets is $\frac{1490\text{bytes}\cdot 8\text{ bits}}{300\text{ kbit/s}} = 39.7$ ms. Thus, ABE packets cannot meet previous ABE packets in the queue so that the queue threshold $T_q = 1$ saves them from being dropped due to a passed deadline. Hence, dropping is turned off for ABE traffic and DSCD behaves like a FIFO queue. This is different for LogNormal-distributed inter-arrival times where $T_q = 1$ cannot prevent excessive packet loss that effectively (see Section V-C2). For an ABE traffic rate of $R_{ABE} = 1$ Mbit/s, the inter-arrival time of ABE packets is $\frac{1490\text{bytes}\cdot 8\text{ bits}}{1\text{ Mbit/s}} = 11.92$ ms. Thus, if an ABE packet arrives and meets another ABE packet, that packet will be dropped as it is 11.92 ms old and has exceeded any of the considered delay thresholds $T_d \in \{2, 5, 10\}$ ms. For ABE traffic rates of $R_{ABE} = 3$ Mbit/s or larger, the inter-arrival time of the packets is $\frac{1490\text{bytes}\cdot 8\text{ bits}}{3\text{ Mbit/s}} = 3.58$ ms or smaller. This is short enough so that delay thresholds T_d have an impact on packet loss and lead to different system behaviour. The number of TCP flows influences the congestion level which has also an impact on the packet loss. We observe packet loss values between 0.5% and 1.4% depending on the specific setting. While the number of TCP flows has a non-monotonic impact on packet loss, smaller delay thresholds lead to more packet loss.

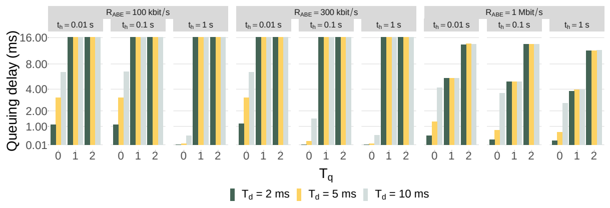
The behaviour of the queuing delay in Figure 11(a) is roughly inverse to the packet loss. For $R_{ABE} = 300$ kbit/s, the queuing delay is about 16 ms which is about the same as for BE traffic. It is lower for $R_{ABE} = 1$ Mbit/s, but it is the same for the different delay thresholds T_d. And for larger ABE traffic rates, the queuing delay clearly decreases with the delay threshold. The number of TCP flows has only a minor impact on the queuing delay of ABE traffic.

2) *Impact of the Queue Threshold T_q:* Algorithm 2 utilizes a queue threshold T_q to prevent excessive packet loss for low ABE traffic rates. We show that $T_q = 1$ is the appropriate parameter.

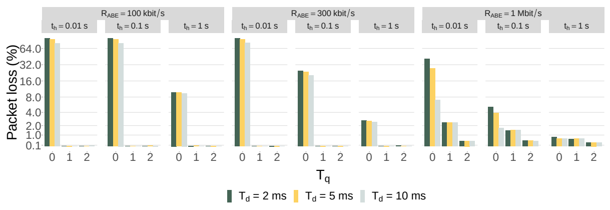
We consider various low rates R_{ABE} of periodic ABE traffic and 64 TCP Cubic background flows. We study different half-life times t_h, delay thresholds T_d, and queue thresholds T_q. To obtain reliable results for $R_{ABE} \in \{100, 300\}$ kbit/s, we extend the data collection time to 280 s. Figures 12(a) and 12(b) compile results for packet loss and delay.

In Figure 12(b), we observe for a queue threshold of $T_q = 0$ very high packet loss which is almost the same for any delay threshold T_d. Only for $R_{ABE} = 1$ Mbit/s and $T_q = 0$ the packet loss decreases with increasing half-life time t_h. In contrast, a queue threshold of $T_q \in \{1, 2\}$ keeps the packet loss very low for $R_{ABE} \in \{100, 300\}$ kbit/s and to moderate values for $R_{ABE} = 1$ Mbit/s. Thus, $T_q \in \{1, 2\}$ turns off traffic differentiation in the presence of small ABE traffic aggregates, which saves them from excessive packet loss. For ABE traffic rate $R_{ABE} = 1$ Mbit/s, the packet loss for $T_q = 1$ is larger than the one for $T_q = 2$.

We now discuss the queuing delay in Figure 12(a). For $T_q = 0$, queuing delay is low and scales with the delay threshold T_d. However, that is at the expense of excessive packet loss in case of $R_{ABE} \in \{100, 300\}$ kbit/s. For $T_q \in \{1, 2\}$, the



(a) Queuing delay



(b) Packet loss

Fig. 12. Queuing delay and packet loss of periodic ABE traffic in the presence of 64 TCP flows via BE; delay threshold T_d, queue threshold T_q, and half-life time t_h are varying parameters; other parameters are set as in Table IV.

queuing delay is as large as the one of BE traffic as service differentiation is turned off. For $R_{ABE} = 1$ Mbit/s, $T_q = 1$ leads to clearly lower delay than $T_q = 2$ which still turns off service differentiation. Thus, $T_q = 1$ is the appropriate value to save small ABE traffic aggregates from excessive packet loss and enable traffic differentiation for ABE traffic rates of $R_{ABE} = 1$ Mbit/s or larger.

E. The Need for Exponential Decay

Exponential credit decay over time may increase packet loss. Nevertheless, it is helpful for several reasons. First, without exponential decay, a selfish user may send a large burst of redundant ABE data to accumulate credit for later use. When then relevant ABE data is transmitted, it can be sent with low delay thanks to stored credit. Exponential decay of stored credit largely removes the incentive for this selfish behavior. Second, the sum of credits in the system is limited (see Section III), which may lead to packet drop at enqueue. Therefore, stored, unused credit essentially shortens the queue and can cause packet loss although the physical queue is not full. Credit decay frees the system from stored credit over time and thereby extends the queue capacity towards normal. Third, adaptive protocols such as TCP may benefit from shorter delay of the ABE class under some conditions. Then, TCP flows over ABE may achieve a larger goodput than TCP flows over BE due to shorter perceived RTTs. As a consequence, ABE traffic may suppress BE traffic. However, a design goal of ABE is to avoid that. We show that exponential decay helps to achieve that goal.

We perform the following experiment. A single TCP Cubic flow via ABE competes against multiple TCP Cubic flows via BE. We measure the ABE flow's relative goodput compared to the average goodput of the BE flows for different half-life times t_h and for different RTTs. Table V shows the results. A relative goodput above 100% indicates that ABE has an unfair bandwidth share.

TABLE V

Goodput of a single TCP flow carried over ABE relative to the average goodput of multiple TCP flows carried over BE. Cubic is used as TCP variant; the half-life time t_h, the number of BE flows, and the RTT are parameters.

RTT	#BE flows	t_h			
		10 ms	100 ms	1 s	∞
10 ms	16	2%	99%	167%	201%
	32	2%	129%	198%	226%
	64	4%	157%	244%	267%
	128	6%	164%	281%	329%
30 ms	16	1%	8%	77%	114%
	32	1%	4%	105%	148%
	64	2%	5%	115%	187%
	128	4%	10%	114%	189%

Without credit decay ($t_h = \infty$), the ABE flow takes a clearly unfair traffic share between 114% and 329%. It is larger for a RTT of 10 ms than for a RTT of 30 ms, and it increases with an increasing number of BE flows. For comparison, $t_h = 1$ s causes relative goodputs between 77% and 281%. For $t_h = 100$ ms the relative goodputs are between 99% and 164% in case of a very low RTT of 10 ms, and between 4% and 10% for larger RTT. When the half-life time is too short ($t_h = 10$ ms), the ABE flow achieves only little goodput (< 6%) as credit decays so fast that subsequent packets cannot profit from it sufficiently. Thus, a half-life time of $t_h = 100$ ms limits the unfairness caused by TCP to a moderate degree and leads only to moderate packet loss for ABE traffic (see Section V-D). Therefore, $t_h = 100$ ms is a preferred configuration value for the half-life time.

F. Inter-Protocol and Inter-Class Unfairness of TCP Variants

There is a large number of TCP variants which do not necessarily share bandwidth in a fair manner as they implement different congestion control algorithms. We call this inter-protocol unfairness. In Section V-E we have already shown that flows with the same TCP variant can share bandwidth in an unfair manner when carrying traffic over both ABE and BE. We call this inter-class unfairness. In the following, we first quantify the inter-protocol unfairness between TCP Cubic and TCP BBR. Then we investigate the inter-class unfairness of both TCP variants separately under various networking conditions. We use the default parameters of Table IV in all experiments.

1) Inter-Protocol Unfairness between TCP Cubic and TCP BBR: Inter-protocol unfairness is a well-known phenomenon [25], [26]. We illustrate it in the following experiment. An equal number of TCP Cubic and TCP BBR flows is carried over a single link and we vary the number of flows and the RTT. All traffic is carried over BE. We take the relative goodput of TCP BBR vs. TCP Cubic as a measure of unfairness. Figure 13 shows the results. For low RTT (10 ms), the goodput of BBR is about 3 times the goodput of Cubic. The number of flows has only a secondary impact. For larger RTT (30 ms and 100 ms), the goodput of BBR is 30–100 times larger than the one of Cubic. Thus, inter-protocol unfairness of existing TCP variants can be enormous.

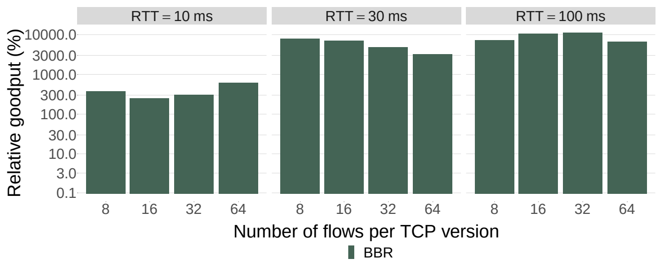
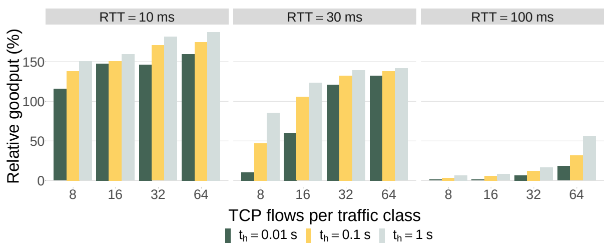
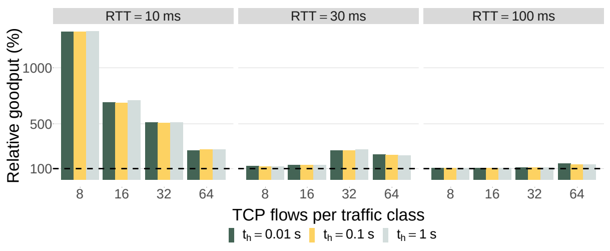


Fig. 13. Goodput of TCP BBR flows relative to the goodput of TCP Cubic flows when being carried over BE; TCP BBR and TCP Cubic have the same number of flows which is a varying parameter as well as the RTT; other parameters are set as in Table IV.

2) Inter-Class Unfairness with TCP Cubic: To quantify inter-class unfairness, we transmit the same number of TCP Cubic flows via ABE and via BE in the system. Apart from that, the experiment setup is the same as before [9]. Figure 14(a) shows the relative goodput for TCP Cubic via ABE vs. TCP Cubic via BE.



(a) Cubic vs. Cubic



(b) BBR vs. BBR

Fig. 14. Goodput of TCP flows via ABE relative to goodput of TCP flows via BE. The experiment is carried out for TCP Cubic and TCP BBR; BE and ABE carry the the same number of flows which is a varying parameter as well as the RTT; other parameters are set as in Table IV.

For an RTT of 10 ms, the relative goodput of ABE vs. BE is between 100% and 185%. The unfairness increases with the number of flows in the system and with increasing half-life time t_h. It is significantly lower than the inter-protocol unfairness between TCP Cubic and TCP BBR for the same RTT. For an RTT of 30 ms, the relative goodput decreases and is clearly below 100% if the number of competing flows is low. For large RTT of 100 ms, the relative goodput is generally below 100%, i.e., TCP senders cannot obtain an unfair traffic share when transmitting over ABE. This is an interesting result as inter-class unfairness is a particular issue at short RTTs

[9]This experiment is slightly different than the similar experiment series in Section V-D where a single ABE flow competes against multiple BE flows.

while inter-protocol unfairness is a particular issue at longer RTTs (see Section V-F1).

We argue why the inter-class unfairness occurs and why the behavior depends on the RTT. Cubic adapts its congestion window based on a cubic function and is mainly influenced by its experienced packet loss. While the congestion window growth of Cubic is independent of the RTT, it still relies on the RTT for timeout calculation. The timeout implicitly affects a parameter for the congestion window growth function. ABE flows experience a relatively lower end-to-end delay (RTT + queuing delay) than BE flows resulting in a higher goodput. This has also been shown in [27], where a smaller RTT leads to higher throughput compared to other Cubic flows with higher RTT. The relative delay advantage vanishes with higher RTTs.

3) Inter-Class Unfairness with TCP BBR: We now look at the inter-class unfairness with TCP BBR. We conducted similar experiments whose results are compiled in Figure 14(b). For small RTT of 10 ms, the relative goodput for ABE flows is between 200% and 1400% depending on the number of flows. The impact of the half-life time t_h is low. Increasing the RTT leads to lower relative goodputs for ABE flows between 100% and 200%. This is a different behaviour than with TCP Cubic. Thus, in case of a predominant deployment of TCP BBR, ABE BBR flows could partly suppress BE BBR flows. However, the problem of BBR suppressing other TCP variants in the current BE Internet is larger and shows that too aggressive congestion control algorithms can be problematic.

The reason why BBR benefits so much from ABE, even at large RTTs, is that its congestion control algorithm does not react to packet loss, which is unlike TCP Cubic. It rather reduces its transmission rate when it notices an increase in the RTT [28]. As, BBR flows via ABE see shorter and more stable end-to-end RTTs due to less queueing delay, they benefit from ABE at any RTT and do not suffer too much from experienced packet loss. The behavior of BBR shows that concepts such as RTT-fairness and influence of AQMs must be considered in the design of congestion control algorithms. As ABE is primarily designed for realtime traffic – and therefore UDP – ABE may be limited to UDP traffic to prevent ABE BBR from suppressing other BE BBR flows. However, this will not work for QUIC-based transport protocols.

VI. Summary and Discussion

We summarize this work and discuss the findings.

A. Novelties of DSCD

The objectives of DSCD are similar to those of DSD and DSF but its properties differ in important aspects.

(1) DSCD has only moderate complexity. A Linux kernel implementation demonstrates its feasibility of 100 Gbit/s links.

(2) DSCD copes with unknown and varying bandwidth while DSD and DSF require a static link bandwidth C for deadline computation. DSCD also measures the link bandwidth C but needs it only to drain credits in the absence of congestion, which is a rather uncritical process.

(3) The conception of ABE is problematic for low rates of ABE traffic. If a packet is dropped due to exceeded delay, there may be no subsequent packet that could leverage that loss for an delay advantage when the queue has been flushed by the next packet arrival. Therefore, ABE traffic aggregates may experience large packet loss with other scheduling algorithms. DSCD prevents this by dropping ABE packets only if there are also other ABE packets in the queue, which essentially turns off service differentiation at low ABE traffic rates.

(4) With DSCD, stored credit decays exponentially over time with half-life time t_h. This avoids that credit can be stored arbitrarily long during a congestion phase. It avoids incentives for selfish users to send more traffic than needed.

(5) Existing algorithms spent lots of effort to pursue approximate fairness for flows sent over BE and ABE. We intend ABE primarily for realtime traffic and not for bulk traffic. Therefore, TCP over ABE may obtain a worse service than TCP over BE. Our objective is even a worse service for TCP over ABE because TCP over ABE should not be able to suppress TCP over BE in the same network. DSCD's exponential decay for stored credit helps to achieve that goal.

B. Performance

We tested DSCD scheduling for BE and ABE traffic using non-responsive traffic with bursts, periodic and TCP traffic, as well as TCP traffic with different variants. We showed that the delay threshold T_d controls the queuing delay for ABE traffic. We recommend to set it to $T_d = 10$ ms as lower values lead to larger packet loss. The queue threshold T_q controls the packet loss and turns of service differentiation in the presence of low ABE traffic rates that are smaller than 1 Mbit/s. The experimental results show that $T_q = 1$ is a good value. The half-life time controls how long credit can be stored so that packet loss and delay decrease with increasing half-life time t_h. If it is too large, then TCP over ABE can obtain significantly larger goodput than TCP over BE under some conditions. Setting $t_h = 100$ ms leads to moderate packet loss and only little inter-class unfairness.

Finally, we quantified inter-protocol and inter-class unfairness (see Section V-F) for multiple scenarios. TCP BBR flows can suppress TCP Cubic flows when being carried over BE, in particular for long RTTs. TCP Cubic flows via ABE can suppress TCP Cubic flows via BE, in particular for short RTTs and the problem vanishes for long RTTs. TCP BBR flows via ABE can suppress TCP BBR flows via BE, also in particular for short RTTs. For long RTTs the advantage diminishes but does not fully disappear. This is mainly the problem of BBR's congestion control as it also causes the observed inter-protocol unfairness.

VII. Conclusion

Alternative Best Effort (ABE) is an alternative traffic class for the Internet. ABE traffic experiences shorter delay than Best Effort (BE) traffic at the expense of more packet loss. This must be achieved without delaying and dropping BE traffic compared to the transmission of the entire traffic with a single FIFO queue.

In this work, we addressed the fundamental question whether an ABE service class is technically feasible, how it behaves with up to date transport protocols, and whether it can be implemented on modern hardware. To that end, we proposed DSCD as an algorithm for combined scheduling of BE and ABE traffic. We implemented it in the Linux network stack and it is fast enough for 100 Gbit/s links. Side products are an approximation of the exponential function in the kernel, which is useful for moving average computations, and a bandwidth estimation method that works well even at moderate link utilization. We used a virtualized hardware testbed to study the impact of DSCD on packet loss and delay for both BE and ABE traffic under various conditions. ABE traffic faces significantly shorter delay but more packet loss than BE traffic provided that a critical mass of ABE traffic is available (≈ 1 Mbit/s). Otherwise we see approximate FIFO behaviour so that BE and ABE receive a similar service. Under all conditions, the service for BE traffic is not degraded by design. We recommended configuration parameters for DSCD so that packet loss for ABE traffic remains small and that TCP does not get an unfairly large traffic share when sending over ABE.

ABE may be useful for Internet service providers to offer their customers a low-delay traffic class that does not harm other traffic. It may be attractive for net-neutral service differentiation, and it may serve as a bridge towards a low-delay Internet. In future work, DSCD could be implemented with network acceleration techniques such as smart NICs or the Metron platform [29] [30] for higher performance.

Acknowledgement

The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/2-1. The authors alone are responsible for the content of the paper.

References

[1] J. Gettys and K. Nichols, “Bufferbloat: Dark Buffers in the Internet,” *ACM Queue*, vol. 9, no. 11, Nov. 2011.
[2] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool, “The Effects of Packet Loss and Latency on Player Performance in Unreal Tournament 2003,” in *ACM SIGCOMM Workshop on Network and System Support for Games*, 2004.
[3] S. Liu, M. Claypool, A. Kuwahara, J. Scovell, and J. Sherman, “The Effects of Network Latency on Competitive First-Person Shooter Game Players,” in *International Conference on Quality of Multimedia Experience (QoMEX)*, 2021.
[4] A. D. Domenico, G. Perna, M. Trevisan, L. Vassio, and D. Giordano, “A Network Analysis on Cloud Gaming: Stadia, GeForce Now and PSNow,” *MDPI Network*, vol. 1, no. 3, 2021.
[5] X. Zhang, H. Chen, Y. Zhao, Z. Ma, Y. Xu, H. Huang, H. Yin, and D. O. Wu, “Improving Cloud Gaming Experience through Mobile Edge Computing,” *IEEE Wireless Communications*, vol. 26, no. 4, 2019.
[6] A. Wahab, N. Ahmad, M. G. Martini, and J. Schormans, “Subjective Quality Assessment for Cloud Gaming,” *MDPI J*, vol. 4, no. 3, 2021.
[7] “ITU-T Recommendation G.107 : The E-Model, a computational model for use in transmission planning,” ITU, Tech. Rep., 2015.
[8] S. Blake, D. L. Black, M. A. Carlson, E. Davies, Z. Wang, and W. Weiss, “RFC2475: An Architecture for Differentiated Services,” Dec. 1998.
[9] B. D. et al., “RFC3246: An Expedited Forwarding PHB (Per-Hop-Behavior),” Mar. 2002.
[10] P. Hurley and J.-Y. Le Boudec, “The Alternative Best-Effort Service,” https://tools.ietf.org/html/draft-hurley-alternative-best-effort, Jun. 2000.
[11] V. Stocker, G. Smaragdakis, and W. Lehr, “The State of Network Neutrality Regulation,” *ACM SIGCOMM Computer Communication Review*, vol. 50, no. 1, 2020.
[12] P. Hurley, J.-Y. Le Boudec, P. Thiran, and M. Kara, “ABE: Providing a Low-Delay Service within Best Effort,” *IEEE Network Magazine*, vol. 15, no. 3, May 2001.
[13] M. Karsten, D. S. Berger, and J. Schmitt, “Traffic-Driven Implicit Buffer Management - Delay Differentiation without Traffic Contracts,” in *International Teletraffic Congress (ITC)*, Sep. 2016.
[14] J. You, M. Welzl, B. Trammell, M. Kuehlewind, and K. Smith, “Latency Loss Tradeoff PHB Group,” https://tools.ietf.org/html/draft-you-tsvwg-latency-loss-tradeoff, Mar. 2016.
[15] T. Høiland-Jørgensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, “The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm,” RFC 8290, 2018.
[16] G. Ramakrishnan, M. Bhasi, V. Saicharan, L. Monis, S. D. Patil, and M. P. Tahiliani, “FQ-PIE Queue Discipline in the Linux Kernel: Design, Implementation and Challenges,” in *IEEE Conference on Local Computer Networks (LCN)*, 2019.
[17] T. Høiland-Jørgensen, D. Täht, and J. Morton, “Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways,” in *IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2018.
[18] M. Podlesny and S. Gorinsky, “RD Network Services: Differentiation through Performance Incentives,” in *ACM SIGCOMM*, Aug. 2008.
[19] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, “Queues Don’t Matter When You Can JUMP Them!” in *USENIX Syposium on Networked Systems Design & Implementation (NSDI)*, 2015.
[20] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, I.-J. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl, “Reducing Internet Latency: A Survey of Techniques and Their Merits,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, 2016.
[21] M. Menth and F. Hauser, “On Moving Averages, Histograms and Time-DependentRates for Online Measurement,” in *International Conference on Performance Engineering (ICPE)*, 2017.
[22] “QDisc: Token Bucket Filter.” [Online]. Available: https://man7.org/linux/man-pages/man8/tc-tbf.8.html
[23] S. Hemminger, “Network emulation with NetEm,” *Linux Conf Au*, vol. 844, 2005.
[24] iperf3 team, “iperf3.” [Online]. Available: http://software.es.net/iperf/
[25] M. Hock, R. Bless, and M. Zitterbart, “Experimental Evaluation of BBR Congestion Control,” in *IEEE International Conference on Network Protocols (ICNP)*, 2017.
[26] Y. Cao, A. Jain, K. Sharma, A. Balasubramanian, and A. Gandhi, “When to Use and When Not to Use BBR: An Empirical Analysis and Evaluation Study,” in *Internet Measurement Conference*, 2019.
[27] T. Kozu, Y. Akiyama, and S. Yamaguchi, “Improving RTT Fairness on CUBIC TCP,” in *International Symposium on Computing and Networking*, 2013.
[28] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “BBR: Congestion-Based Congestion Control,” *ACM Queue*, vol. 14, no. 5, Sep. 2016.
[29] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr., “Metron: NFV Service Chains at the True Speed of the Underlying Hardware,” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.
[30] G. P. Katsikas, T. Barbette, D. Kostić, J. G. Q. Maguire, and R. Steinert, “Metron: High-Performance NFV Service Chaining Even in the Presence of Blackboxes,” *ACM Transactions on Computer Systems*, vol. 38, no. 1–2, 2021.

Steffen Lindner is a Ph.D. student at the chair of communication networks of Prof. Dr. habil. Michael Menth at the Eberhard Karls University Tuebingen, Germany. He obtained his master’s degree in 2019 and afterwards, became part of the communication networks research group. His research interests include software-defined networking, P4 and congestion management.

Gabriel Paradzik is a Ph.D. student at the Eberhard Karls University Tuebingen, Germany. He started his Ph.D. in April 2021 at the communication networks research group. His research interests include congestion management and data center networking.

Michael Menth, (Senior Member, IEEE) is professor at the Department of Computer Science at the University of Tuebingen/Germany and chairholder of Communication Networks since 2010. He studied, worked, and obtained diploma (1998), PhD (2004), and habilitation (2010) degrees at the universities of Austin/Texas, Ulm/Germany, and Wuerzburg/Germany. His special interests are performance analysis and optimization of communication networks, resilience and routing issues, as well as resource and congestion management. His recent research focus is on network softwarization, in particular P4-based data plane programming, Time-Sensitive Networking (TSN), Internet of Things, and Internet protocols. Dr. Menth contributes to standardization bodies, notably to the IETF.

1.4 Learning Multicast Patterns for Efficient BIER Forwarding with P4

Learning Multicast Patterns for Efficient BIER Forwarding with P4

Steffen Lindner, Daniel Merling, Michael Menth
Chair of Communication Networks, University of Tuebingen, Tuebingen, Germany
{steffen.lindner, daniel.merling, menth}@uni-tuebingen.de

***Abstract*—**

Bit Index Explicit Replication (BIER) is an efficient domain-based transport mechanism for IP multicast (IPMC) that indicates receivers of a packet through a bitstring in the packet header. Recently, BIER forwarding has been implemented on 100 Gbit/s per port hardware using the P4 programming language. However, the implementation requires packet recirculation to iteratively serve one next-hop after another. The objective of this paper is to reduce this inefficiency.

Static multicast groups can be configured on P4 switches so that traffic can be sent to all next-hops without recirculation. We leverage that feature to make BIER forwarding more efficient. However, only a limited number of static multicast groups can be configured on a switch, which is not sufficient to cover all potential port patterns. In a first step, we develop efficient BIER forwarding that utilizes static multicast groups derived from so-called configured port clusters. Then, we design port clustering algorithms that observe multicast patterns and compute configured port clusters which are more efficient than randomly selected port clusters. These methods are based on Spectral Clustering, an unsupervised machine learning technique. We perform simulations that underline the effectiveness of this approach to reduce inefficient packet recirculations. We further implement the new forwarding behaviour on programmable hardware and provide a controller that samples BIER packets on the switch, runs the port clustering algorithms, and updates the configured static multicast groups. We validate this open source implementation in a testbed and show that the experimental results are in line with the simulation results.

***Index Terms*—Software-Defined Networking, Bit Index Explicit Replication, Multicast, Resilience, Scalability, Unsupervised Machine Learning**

I. INTRODUCTION

IP multicast (IPMC) is an efficient way to distribute one-to-many traffic. It is organized into multicast groups that are identified by unique IP addresses. Traffic of a multicast group is sent to all subscribers along a distribution tree, i.e., nodes replicate and forward packets to specific neighbors towards the subscribers. Therefore, only one packet is sent over each involved link, which reduces the load in comparison to unicast. To that end, core nodes store for each multicast group the neighbours that should receive a packet copy. As a result, traditional IPMC has two scalability issues. First, whenever the composition of an IPMC group changes, signaling to core nodes is necessary to update the neighbors that should receive packet copies. Second, link or node failures, and topology changes may affect multiple multicast groups, which puts high signaling and processing load on core devices.

The IETF proposed Bit Index Explicit Replication (BIER) [1] as an efficient and stateless domain-specific transport mechanism for IPMC traffic. Ingress routers equip an IPMC packet with a bitstring in the BIER header which contains all destinations of the packet within the domain. Core nodes replicate and forward the BIER packet according to its bitstring and the paths from the interior gateway protocol (IGP) which is called routing underlay. Egress routers remove the BIER header, and IPMC processing continues. With BIER, only ingress and egress routers of a domain know IPMC groups and are involved in signalling, but not the core routers.

Recently, we presented an open source BIER implementation for 100 Gbit/s per port in P4 for the Tofino ASIC hardware [2]. This implementation is inefficient as it requires one pipeline iteration per next-hop of a BIER packet as packets are transmitted iteratively instead of simultaneously. Therefore, a packet with n next-hops requires $n-1$ recirculations. On the one hand this is due to the fact that packet replication to a dynamic set of outgoing ports is not supported on the specific hardware device. On the other hand, it is difficult[1] to derive the set of outgoing ports from the bitstring within a single pipeline iteration, which is a general challenge for all switch architectures.

In this paper we present an efficient BIER implementation in P4 for the Tofino ASIC. First, we propose a forwarding algorithm that utilizes static multicast groups to simultaneously forward BIER packets to many outgoing ports. However, the number of configurable static multicast groups is limited and does not suffice to cover all port combinations on a 32-port switch. Therefore, the algorithm leverages subsets of ports, so-called "configured" port clusters. All port combinations within a configured port cluster are configured as static multicast groups. This allows efficient BIER forwarding within a very few pipeline iterations (at most 3 or 4 on the Tofino). To further improve the efficiency, we suggest to choose configured port clusters such that they contain ports over which BIER packets are frequently forwarded together. To that end, we propose port clustering algorithms that learn port patterns from sampled BIER traffic and compute configured port clusters that reduce the number of required forwarding cycles. The methods

The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/3-1. The authors alone are responsible for the content of the paper.

[1]BIER bitstrings consist of at least 256 bits and each of them identifies a receiver. This results in 2^{256} possible bit combinations which need to be mapped to up to 2^{32} outgoing port combinations on a switch with 32 ports. This is a challenge for naive table matching.

are based on Spectral Clustering which is an unsupervised machine-learning technique. In practice, a controller applies port clustering from time to time on recently sampled BIER traffic and updates the configured port clusters on the switch.

The paper is structured as follows. In Section II and III we describe related work and give an introduction to Bit Index Explicit Replication (BIER). Sections IV and V give a primer on the programming language P4 and cover important aspects of the existing P4-based BIER implementation. Section VI proposes the efficient BIER forwarding algorithm and shows simulation results for arbitrarily selected configured port clusters. Section VII suggests various port clustering methods. Their performance is compared by simulation in Section VIII and by hardware experiments in Section IX. Finally, we conclude the paper in Section X.

II. Related Work

In this section we first review related work on traditional multicast and resilience. Then, we present work related to both SDN- and BIER-based multicast. Finally, we review clustering approaches.

A. Traditional Multicast

Islam et al. [3] and Al-Saeed et al. [4] investigate related work for traditional multicast. The majority of cited papers aim to improve the scalability of traditional IPMC. They present intelligent tree-building mechanisms for multicast to make it more efficient, e.g., by reducing required state, or signaling.

Elmo [5] encodes topology information of data centers in packet headers to improve the scalability of IPMC. It leverages characteristic properties of those topologies to reduce the size of the forwaring information base (FIB) of core routers. The Avalanche Routing Algorithm (AvRA) [6] follows a similar approach where it optimizes link utilization for multicast by leveraging topology characteristics of data center networks. Dual-Structure Multicast (DuSM) [7] separates forwarding structures for high-bandwidth and low-bandwidth traffic to improve scalability and link utilization in data centers. Li et al. [8] optimize the FIB to improve the scalability of traditional multicast in data center networks. To that end, they propose to partition the multicast address space and aggregate those at bootleneck switches.

Application layer multicast (ALM) [9] monitors the traffic on application-specific distribution trees to optimize their structures for the corresponding group objective. Mokhtarian et al. [10] construct minimum-delay trees to reduce latency for delay sensitive data with different requirements like min-average, min-maximum, real-time requirements, etc. Adaptive SDN-based SVC multicast (ASCast) [11] follows a similar approach. The authors describe an integer-linear program to build optimal distribution trees and fast forwarding tables to optimize multicast forwarding in terms of latency and delay for live streaming.

Kaafar et al. [12] present a building scheme for efficient overlay multicast trees based on location-information of subscribers. Boivie et al. [13] propose small group multicast (SGM) which aims at avoiding management and set up overhead for multicast groups with a small number of receivers. To that end, the multicast packets of such groups carry the distribution information in their headers, which avoids signaling in the core. Simple explicit multicast (SEM) [14] stores multicast information only on branching nodes of the distribution tree. Non-branching nodes forward packets to the next-branching node via unicast. Jia et. al. [15] leverage prime numbers and the Chinese remainder theorem to efficiently organize the FIB. They reduce the size of the FIB in core devices and facilitate implementation.

Steiner trees [16] are tree structures that are used to build efficient multicast trees. Many research papers modify Steiner trees to build multicast trees optimized with regard to a specific metric, e.g., link costs [17], delay [18], number of hops [19], number of branch nodes [20], retransmission efficiency [21], or optimal placement of IPMC sources [22].

B. Resilience for Multicast

Shen et al. [21] extend Steiner trees so that distribution trees contain recovery nodes. Such nodes cache multicast traffic for retransmission to cut off receivers after recomputation of the FIB. The authors of [23] investigate resilience of several multicast algorithms against node failures. Kotani et al. [24] deploy primary and backup multicast trees that are identified by a field in the packet header. After failure detection, the source sends its packet over a working backup tree by indicating the backup path in the packet header. Pfeiffenberger et al. [25] propose that each node in a distribution tree is also the root of a backup tree that reaches all downstream destinations over paths that do not include the failed link/node. Nodes switch packets on a backup tree by setting a VLAN tag in the packet header.

C. SDN-Based Multicast

Rückert et al. [26], [27] propose and extend Software-Defined Multicast (SDM) which is an OpenFlow-based multicast platform to facilitate management. It focuses on overlay-based live streaming services for P2P video live streaming. The authors of [28] describe address translation in OpenFlow switches to reduce the number of multicast-dependent forwarding entries in near-to-leaf nodes. To that end, the forwarding action from the last hop towards the receivers is done with a unicast address. Lin et al. [29] implement shared multicast trees between different IPMC groups on OpenFlow switches. Thereby, the number of forwarding entries is reduced. The authors of [30] leverage bloom filters to reduce the number of TCAM-entries that is required for SDN-based multicast.

D. BIER Multicast

In [31], [32] we presented an early prototype of a BIER implementation in P4 for the software switch bmv2 [33]. However, bmv2 yields only low throughput (900 Mbit/s) [34]. Therefore, we developed a P4 implementation of BIER and BIER-FRR for the P4-programmable switching ASIC Tofino [2] with a switching capacity of 3.2 Tb/s, i.e., 100 Gbit/s per port in a 32-port switch. We demonstrated its technical feasibility and performance limits.

Giorgetti et al. [35], [36] presented an OpenFlow implementation of BIER. However, it requires extensive state or controller interaction for efficient BIER forwarding. Furthermore, it is capable of addressing only 20 receivers per packet due do the limited size of MPLS labels which are used to implement arbitrary header fields.

Desmouceaux et al. [37] investigate the retransmission efficiency of BIER. That is, when subscribers signal missing packets, BIER allows to retransmit packets to only specific subscribers while still forwarding only one packet copy per link. Traditional multicast retransmits either via unicast or to the entire multicast group. The evaluations show that BIER is significantly more efficient than traditional multicast, i.e., it causes fewer retransmitted packets and achieves better link utilization.

BIER with tree engineering (BIER-TE) [38] encodes the entire distribution tree in the packet header to have more control of the paths. Carrier grade minimalist multicast (CGM2) [39] is a novel derivate of BIER-TE. It encodes the distribution tree in a recursive manner in the packet header. Thereby, it can scale to larger networks than BIER-TE. However, CGM2 has not been implemented, yet, and is still under development.

Braun et al. [40] propose 1+1 protection for BIER where traffic is transported on two disjoint trees. As a result, traffic is delivered successfully to receivers even when a failure interrupts one tree.

The state of the art for BIER multicast with P4 is limited due to the following reasons. First, existing implementations require additional forwarding capacity as shown in [2] which may significantly reduce the usable physical ports of a switch. Second, other BIER implementations require either exponential state or significant controller interaction (as in native IP multicast), which is contrary to the stateless nature of BIER. In this work, we present a novel approach for efficient BIER forwarding that leverages static multicast groups to reduce the required forwarding capacity by eliminating excessive recirculation. Further, we optimize the selection of the static multicast groups with unsupervised machine learning to further reduce the required recirculation and therefore forwarding capacity.

E. Clustering

Clustering is an unsupervised machine learning technique that solves the problem of identifying clusters of data points in a multidimensional space. Given a set of D-dimensional points $\{x_1, ..., x_N\}$, the goal of clustering is to partition the data into groups/clusters such that points in the same cluster are similar and points in different groups are dissimilar.

k-Means [41] is one of the most applied clustering algorithms. Its incentive is to find an assignment of data points to k cluster centers such that the sum of the squares of the distances of each data point to its cluster center is minimized.

DBSCAN [42] is a density-based clustering algorithm that can form arbitrary clusters and is especially suited for outlier detection. It is not suited for high-dimensional data sets as it uses the euclidean distance as similarity measure.

Spectral Clustering [43] is a clustering algorithm that is based on graph properties. It uses the normalized Laplacian[2] of the similarity matrix of the data points to build k clusters. Data points are embedded in $\mathbb{R}^k$ through the so-called spectral embedding. Thereby, the first k eigenvectors of the Laplacian are computed and used to project the data points. Finally, the embedded data points are clustered with a simple clustering algorithm, e.g, k-Means.

III. Bit Index Explicit Replication (BIER)

In this section we give a short primer on BIER. BIER is a domain-based transport mechanism for multicast traffic. It can be explained with three layers as shown in Figure 1. On

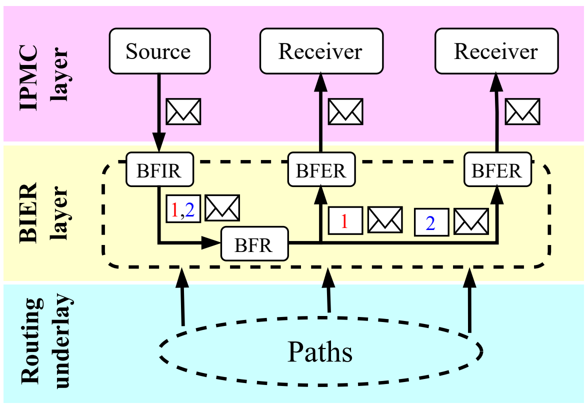


Figure 1: Layered BIER architecture according to [32].

the IPMC layer sources and receivers send and receive IPMC packets. The BIER layer is responsible for the transport of the IPMC packets from the IPMC sources to the IPMC receivers along paths from the unicast routing, i.e., the routing underlay, through the so-called BIER domain.

The BIER domain consists of three types of BIER devices. First, bit forwarding ingress routers (BFIRs) are the entry points to the BIER domain. They encapsulate IPMC packets with a BIER header for forwarding within the BIER domain. The BIER header contains a bit string that indicates all destinations of the BIER packet. That is, each bit position corresponds to a specific destination. A bit is activated if the corresponding destination should receive a copy of the packet. Second, bit forwarding routers (BFRs) forward BIER packets towards their destinations according to the activated bits in the BIER header. That is, a BFR sends a packet copy to the first next-hop over which at least one destination is reached. It leaves only those bits activated in the bit string of the packet copy which correspond to destinations that are reached via that next-hop, and clears all other bits to prevent duplicates at the receivers. The BFR repeats this procedure until all destinations are served. As a result, the forwarding path of a BIER packet is a tree whose links carry only a single packet copy. Third, bit forwarding egress routers (BFERs) remove the BIER header and pass the IPMC packet to the IPMC layer.

[2]The normalized Laplacian of a graph with weight matrix W and degree matrix D is given as $L = D^{-1/2}(D - W)D^{-1/2}$ [43].

Next-hops on the BIER distribution tree may not be reachable due to link or node failures. In this case, downstream destination nodes do not receive any BIER traffic until BIER forwarding tables are updated. Therefore, two BIER fast reroute (BIER-FRR) concepts have been proposed [44] to forward BIER traffic over backup paths from the detection of the failure until BIER forwarding tables have been updated. The methods have been compared in [45] and tunnel-based BIER-FRR has been implemented in [2].

IV. Introduction to P4

In this section we give an overview of P4, explain the P4 processing pipeline, packet cloning, packet recirculation, and multicast groups in P4.

A. P4 Overview

P4 (programming protocol-independent packet processors) [46] is a high-level programming language to describe the data plane of P4-programmable devices. It is applied in a wide range of applications and research [47]. Target-specific compilers map the P4 programs to the programmable processing pipeline of the target devices which are also called targets. The P4 compiler also generates a data plane API that can be used by a control plane to manage runtime behavior, e.g., by writing forwarding entries. P4 targets follow a certain architecture that may vary between different targets, e.g., Intel Tofino implements the TNA architecture whereas some P4 capable SmartNICs may implement the PSA architecture. Packet cloning, multicast, and recirculation are common features that are supported by most P4 architectures. We implemented the subsequently presented mechanism for the Intel Tofino, which follows the TNA architecture. Therefore, most explained P4 related concepts are done with the TNA in mind. However, the presented concepts and mechanisms can be implemented similarly in other architectures.

B. P4 Pipeline

Figure 2 shows the abstract pipeline model of a P4 programmable device [46]. A programmable parser deserializes

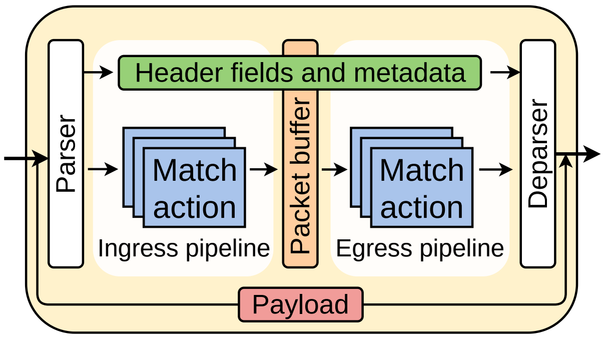


Figure 2: Abstract forwarding model according to [46].

the packet header and stores the information in so-called header fields. The header fields are carried through the pipeline together with packet-specific metadata fields which are comparable to variables from other high-level programming languages. Only header fields and metadata are processed afterwards in the ingress pipeline, i.e., the payload of the packet remains untouched. The ingress pipeline consists of one or more match-action-tables (MATs) that map header fields or metadata to actions. Examples for actions are changing header fields or metadata, e.g., setting the egress port of the packet. After processing in the ingress pipeline, the packet is temporarily buffered so that it can be processed by the egress pipeline which works similarly to the ingress pipeline. Finally, the deparser serializes the possibly changed header fields, forwards the packet through the designated egress port, and discards the metadata.

C. Packet Cloning

P4 has an operation for packet cloning. Depending on the architecture, different clone operations are defined. In the following, we explain ingress to egress (I2E) cloning which we used for the implementation. With I2E cloning, a set metadata flag indicates that the packet should be cloned after its processing in the ingress pipeline has concluded. However, the header fields and metadata of the clone resemble the packet that is initially parsed before the ingress pipeline. Figure 3 shows the concept. After the ingress pipeline has finished,

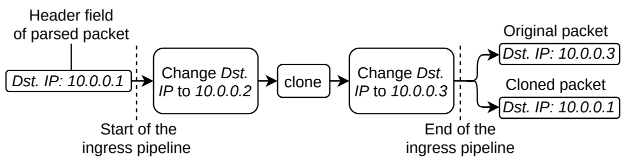

Header field of parsed packet
Dst. IP: 10.0.0.1
Start of the ingress pipeline
Change Dst. IP to 10.0.0.2
clone
Change Dst. IP to 10.0.0.3
End of the ingress pipeline
Original packet
Dst. IP: 10.0.0.3
Cloned packet
Dst. IP: 10.0.0.1

Figure 3: When a packet is cloned, the copy is created only after the ingress pipeline and its header fields are reset to the initial value after the packet has been parsed.

the packet is cloned and both the original packet, i.e., with header changes, and the packet copy, i.e., without header changes, enter the egress pipeline where they are processed independently of each other. Some architectures allow to carry additional information during cloning. In the case of the TNA, this is done through a so-called mirror header.

D. Packet Recirculation

Packet recirculation in P4 allows a packet to be processed a second time by the entire pipeline, i.e., by ingress and egress pipeline. To recirculate a packet, its egress port, i.e., a special metadata field, is set to a particular port ID that corresponds to a switch-intern recirculation port. The recirculation port functions as a regular port of the switch with the exception that it has no physical connector, i.e., only the switch itself can send to and receive traffic from the recirculation port.

After the packet has been processed by both the ingress and egress pipeline, it is sent to the recirculation port. Afterwards, the packet is processed again as if it has been received on a physical port.

E. Static Multicast Groups

P4 allows controllers to configure multicast groups on forwarding devices. A multicast group consists of a tuple of multicast group identifier and a set of egress ports. In addition, there is a special metadata field that allows the ingress pipeline to assign a multicast group identifier to a packet. After the ingress pipeline has completed, the packet is replicated to the pre-defined set of egress ports.

In the following we refer to those configured multicast groups as "static multicast groups" to differentiate them from multicast groups of IPMC. A static multicast group is a local mechanism on a switch to simultaneously forward a packet to multiple egress ports.

V. Simple P4-Based BIER Implementation

In this section we review the simple P4-based BIER implementation of [2]. The target is the Intel Tofino high-speed switching ASIC [48]. It is used for a prototype on the Edgecore Wedge 100BF-32X [49] with 32 100 Gbit/s ports. The implementation makes heavy use of packet recirculation, which causes capacity issue, e.g., 100 Gb/s incoming multicast traffic with 5 next-hops requires 400 Gb/s additional forwarding capacity for recirculation purpose, i.e., it requires #next-hops - 1 recirculations. The efficient BIER implementation in Section VI builds upon the simple implementation and greatly reduces the need for recirculations.

A. BIER Processing

BFRs leverage the Bit Index Forwarding Table (BIFT) to determine the next-hops of a BIER packet. We implement the BIFT as common match-action table in P4. For each BFER there is one entry in the BIFT. The match key is a bitstring with only the single bit activated for the corresponding BFER. The other entry fields, i.e., the corresponding action with its parameters, are a next-hop and a forwarding bitmask (FBM). The FBM is a bit string similar to the BIER bitstring and it indicates the BFERs with the same next-hop. When a packet arrives, the BFR first copies the bitstring of the packet to a temporary metadata field which we call "remaining bits". The remaining bits indicate the BFERs that still have to be served. Then, the least-significant activated bit in the remaining bits is matched against the BIFT using a ternary match operation. The match-action table entry returns the corresponding next-hop and FBM for that BFER. The BFR clears all bits in the bitstring of the packet that are not activated in the FBM. Thus, only the bits of BFERs that are reached through this next-hop remain in the BIER bitstring. The BFR further clears all bits in the remaining bits that are activated in the FBM as they have already been served. Afterwards, the clone operation is applied. Figure 4 shows the processing flow of the original and cloned BIER packet. The original packet is sent through the appropriate egress port to reach the selected next-hop. The packet copy is cloned to the egress pipeline and recirculated to a recirculation port. Within the egress pipeline, the BIER bitstring of the packet copy is set to the remaining bits so that only the remaining BFERs are served in the next pipeline iteration. Further details about the original BIER forwarding implementation can be found in [2].

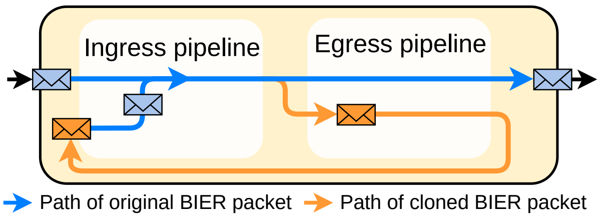


Figure 4: The original BIER packet is sent through an egress port while the packet copy is recirculated.

B. Recirculation Capacity and Problem Statement

The Tofino ASIC has a switch-intern recirculation port which has the same packet processing capacity as regular ports. If its capacity does not suffice, packet loss occurs. To increase the recirculation capacity, physical ports may be turned into loopback mode, and recirculation traffic may be distributed over the internal ports and the loopback ports in a round-robin manner [2]. As these ports cannot be utilized for other traffic, recirculations are costly.

The simple BIER implementation requires $n-1$ recirculations for BIER packets with n next-hops. This approach obviously does not scale well with increasing number of next-hops and traffic rate. The objective of this paper is a more efficient P4-based implementation that requires fewer recirculations per BIER packet (see Section VI) and an optimized configuration thereof using clustering methods (see Section VII).

VI. Efficient BIER Forwarding with P4

We explain how static multicast groups can be leveraged to make BIER forwarding using P4 more efficient, and how BIER-FRR can be integrated. To demonstrate the efficiency of the new forwarding algorithm, we present a simulative performance study.

A. Efficient BIER Forwarding with Static Multicast Groups

We first explain how BIER forwarding can profit from configured port clusters consisting of static multicast groups such that multiple next-hops can be served within a single pipeline iteration. Then we explain how the forwarding algorithm determines a port cluster and the appropriate static multicast group for a BIER packet, and forwards it.

The presented algorithm is specific to P4 and the architecture of the Tofino ASIC. However, efficient forwarding algorithms for any switch architecture need to determine the set of egress ports for a BIER packet. This is a difficult task as bitstrings are at least 256 bits large. Therefore, the presented approach may also be a useful base for efficient BIER forwarding on other switch architectures.

1) Use of Static Multicast Groups: The idea to make BIER forwarding more efficient is the use of static multicast groups so that multiple egress ports can be simultaneously served instead of using recirculation.

A naive solution is configuring static multicast for all possible combinations of egress ports. When a packet arrives, the set of egress ports is determined and the corresponding static multicast group forwards the packet to all needed egress ports without packet recirculation. However, on a 32 port switch this requires $2^{32} = 4294967296$ static multicast groups, which exceeds the number of configurable static multicast ports.

We propose now a more sophisticated approach which requires fewer static multicast groups. We define a set $\mathcal{C} = \{C_1, ..., C_k\}$ of so-called "configured port clusters" (or port sets) C_i such that all configured port clusters together cover all ports of a switch. Configured port clusters may be disjoint or overlapping. For each port set C_i, static multicast groups $M_j \in \mathbb{P}(C_i)$ are configured for all sets of ports in the powerset of C_i. Thus, a configured port cluster C implies

$$m(C) = 2^{|C|} - |C| - 1 \tag{1}$$

static multicast groups that need explicit configuration on the switch; the empty group and groups with only a single destination do not need to be configured. On a 32-port switch three port clusters with 10, 11, and 11 ports may be configured, which requires in total 5085 explicitly configured static multicast groups. This is well feasible on a switch like the Tofino which supports up to $2^{16} = 65536$ static multicast groups[3]. With this approach, a BIER packet needs to be sent to at most $|\mathcal{C}|$ static multicast groups, which requires $|\mathcal{C}| - 1$ recirculations instead of $n_h - 1$ with n_h being the number of next-hops of a BIER packet. Moreover, the administrator may set a threshold m_{max} on the number of static multicast groups usable for efficient BIER forwarding.

2) Forwarding Procedure: We first give a forwarding example. Then, we describe how the forwarding procedure selects a set of configured port clusters $\mathcal{S}_i \subseteq \mathcal{C}$ for BIER forwarding, and then we present how the appropriate static multicast group is chosen from a selected configured port cluster $C_j \in \mathcal{S}_i$.

a) Forwarding Example: Figure 5 illustrates an example for a 8-port switch with three configured port clusters, $\mathcal{C} = \{C_1 = \{1, 2, 3\}, C_2 = \{4, 5, 6\}, C_3 = \{6, 7, 8\}\}$. For each configured port cluster, all port combinations are configured as static multicast groups. The empty group and groups with only a single port do not need to be configured. We consider all subsets of configured port clusters $\mathcal{S}_i \subseteq \mathcal{C}$. A packet destined for ports 1, 3, and 4 requires the subset $\mathcal{S}_4 = \{C_1, C_2\}$ for forwarding, i.e., it will be served by the multicast group $\{1, 3\}$ from C_1 and the multicast group $\{4\}$ from C_2, which requires a single recirculation. Note that groups with a single destination do not need to be configured as static multicast group.

b) Selection of Set of Configured Port Clusters: Now we explain how the appropriate subset $\mathcal{S}_i$ for forwarding is determined in the data plane. C-FBM($\mathcal{S}_i$) is the combined forwarding bitmask of a subset of configured port clusters and indicates all BFERs that are reachable through $\mathcal{S}_i$. We set up a match-action table with one entry per subset $\mathcal{S}_i$ in increasing order with regard to subset size $|\mathcal{S}_i|$, as shown in Figure 5. The entry ¬C-FBM($\mathcal{S}_i$) is the complement of C-FBM($\mathcal{S}_i$). The objective is to find the smallest subset of configured port clusters that serves all BFERs of a BIER packet. To that end, the bitstring of a packet is bitwise ANDed with the complement of the C-FBM in the match-action table. We define a match if the result of that operation is zero. Then, all BFERs of the BIER packet are served by the corresponding subset $\mathcal{S}_i$. This operation is done through a ternary match. A ternary match in P4 is defined by a source value s_v, e.g., a header field, and a ($mask$, $value$) pair. The corresponding table entry matches when s_v & $mask = value$. The source value is given by the BIER bitstring, the mask is the complement of the C-FBM and the value is 0. Due to the order within the match-action table, the first match $\mathcal{S}_i$ is the smallest subset with that property. The first configured port cluster C_j in that subset $\mathcal{S}_i$ is selected for the remainder of the forwarding process.

We consider the example of Figure 5, where 8 BFERs are reachable over ports 1-8. For simplicity, BFER i corresponds to the i-th bit in the BIER bitstring and is reachable over port i, i.e., BFER 1 corresponds to the least significant bit and is reachable over port 1. The C-FBMs for all subsets $\mathcal{S}_i \subseteq \mathcal{C}$ are given in Table 1.

Table 1: Subsets $\mathcal{S}_i \subseteq \mathcal{C}$ with corresponding C-FBMs.

Subset	C-FBM	¬C-FBM
$\mathcal{S}_1 : \{C_1\}$	00000111	11111000
$\mathcal{S}_2 : \{C_2\}$	00111000	11000111
$\mathcal{S}_3 : \{C_3\}$	11100000	00011111
$\mathcal{S}_4 : \{C_1, C_2\}$	00111111	11000000
$\mathcal{S}_5 : \{C_1, C_3\}$	11100111	00011000
$\mathcal{S}_6 : \{C_2, C_3\}$	11111000	00000111
$\mathcal{S}_7 : \{C_1, C_2, C_3\}$	11111111	00000000

Again, we assume a BIER packet to be destined for ports 1, 3, and 4, i.e., towards BFERs 1, 3, and 4 with a bitstring $bs = 00001101$; then only $\mathcal{S}_4$ or $\mathcal{S}_7$ can cover all BFERs of the packet[4]. Due to the order within the match-action table, the first match is $\mathcal{S}_4$ and C_1 is selected for the remainder of the forwarding process.

c) Selection of the Static Multicast Group: Only a single static multicast group M_h of the configured port cluster C_j will be used for forwarding. We now determine that $M_h \in \mathbb{P}(C_j)$ and take a similar approach as in Section VI-A2b for that purpose. We set up a match-action table for C_j which has an entry for any static multicast group $M_h \in \mathbb{P}(C_j)$. The entries are sorted by increasing group size $|M_h|$ and contain the complement of the C-FBM of the corresponding multicast group. Single ports are also considered as static multicast groups although they do not require explicit configuration on the switch. The bitstring of a BIER packet is first ANDed with the C-FBM of the selected configured port cluster C_j. This excludes all BFERs from the bitstring that cannot be served by C_j. The result is bitwise ANDed with the complement of the C-FBM(M_h) of the multicast groups in the table entries. We define a match if the result is zero. This is done with a ternary match operation as with the selection of a configured port cluster. Due to the increasing order of entries in the match-action table, the first match refers to the smallest static

[3]The actual usable number of available resources depends on the program complexity.

[4]This is ensured through the ternary match operation: bs & ¬C-FBM(S_i) == 0.

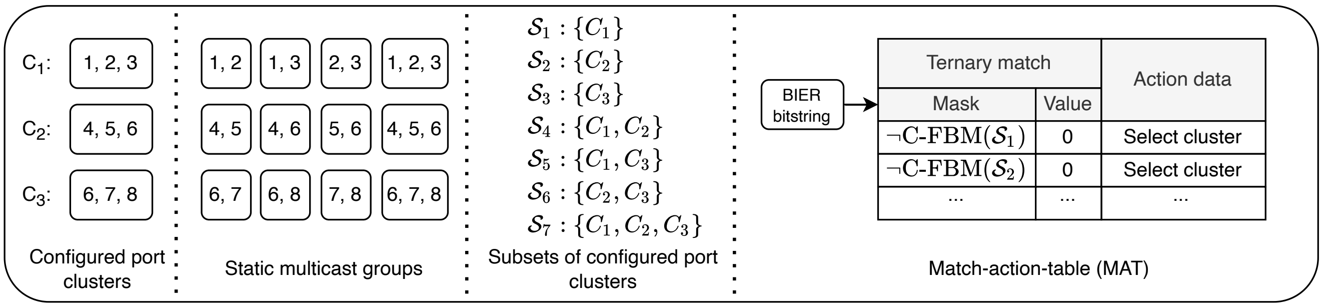

C1: 1, 2, 3
C2: 4, 5, 6
C3: 6, 7, 8
Configured port clusters
1, 2 | 1, 3 | 2, 3 | 1, 2, 3
4, 5 | 4, 6 | 5, 6 | 4, 5, 6
6, 7 | 6, 8 | 7, 8 | 6, 7, 8
Static multicast groups
$\mathcal{S}_1 : \{C_1\}$
$\mathcal{S}_2 : \{C_2\}$
$\mathcal{S}_3 : \{C_3\}$
$\mathcal{S}_4 : \{C_1, C_2\}$
$\mathcal{S}_5 : \{C_1, C_3\}$
$\mathcal{S}_6 : \{C_2, C_3\}$
$\mathcal{S}_7 : \{C_1, C_2, C_3\}$
Subsets of configured port clusters
BIER bitstring
Ternary match
Mask
Value
Action data
$\neg$C-FBM($\mathcal{S}_1$) | 0 | Select cluster
$\neg$C-FBM($\mathcal{S}_2$) | 0 | Select cluster
... | ... | ...
Match-action-table (MAT)

Figure 5: Configured port clusters C_i together cover all ports of a switch. All port combinations within a configured port cluster are configured as static multicast groups. A match-action table chooses a minimum subset of configured port clusters and selects its first configured port cluster for forwarding.

multicast group M_h within the configured port cluster that covers all relevant BFERs.

d) Forwarding and Bitstring Adaptation: At the end of the ingress pipeline, the original BIER bitstring is stored in a transient metadata header. The activated bits in C-FBM(M_h) are deactivated in this transient metadata header which represents the remaining bits that need processing; if the bitstring is not zero, the packet is recirculated and the BIER bitstring is restored through the transient metadata header in the egress pipeline of the recirculation port[5]. In addition, a copy of the original packet is sent to all egress ports of the selected static multicast group M_h. The egress pipelines of these ports clear all bits in the packet's bitstring that are not activated in the FBM of the corresponding port and then they transmit the packets.

B. Integration of BIER-FRR

The proposed efficient forwarding scheme is compatible with BIER-FRR if BIER-FRR is integrated as follows. First, the switch processes the egress ports that are affected by a failure, i.e., a failed link or a failed node. To that end, the BIER packets are forwarded by regular BIER forwarding but over alternate ports. When all affected egress ports have been served, the BIER packet is recirculated and the remaining ports, i.e., working ports, are processed by the presented, efficient forwarding algorithm. This approach prevents duplicates at subscribers and unnecessary double transmissions of the same packet over one link. Details are given in [2].

C. Simulative Performance Evaluation

We evaluate the concept of static multicast groups for efficient BIER forwarding through the following experiment. We examine different numbers of disjoint configured port clusters $k \in \{1, 2, 4, 8, 16, 32\}$. With k configured port clusters and a 32 port switch, each configured port cluster contains $\frac{32}{k}$ ports. Further, we simulate BIER packets with $n_h \in \{1, 2, 4, 8, 16, 32\}$ random next-hops. They are processed by the different configured port clusters. Figure 6(a) and Figure 6(b) show the average number of recirculations per packet and the required static multicast groups. Although multicast traffic with random next-hops is unrealistic (see Section VIII-A1), it serves as a good baseline for a performance evaluation of the efficient BIER forwarding mechanism. The average number of recirculations increases with the number of next-hops n_h and the number of configured port clusters k. In fact, the number of recirculations is bound by $k - 1$. For $k = 32$, the results are equivalent to the simple (original) BIER forwarding. Higher values of k lead to smaller configured port clusters, and hence, to fewer next-hops that can be served in one shot. The number of required static multicast groups decreases with the number of configured port clusters k. To keep the number of recirculations low, larger configured port clusters should be preferred. However, the number of available static multicast groups may be limited due to technical reasons or based on administrative decisions.

In the given traffic model, we randomly selected next-hops for BIER packets. This is not a realistic model for multicast traffic. The next-hops of subsequent BIER packets are likely to be correlated and so are the ports over which the packets are sent. Therefore, some configured port clusters reduce the average recirculation more than others. To effectively minimize the number of recirculations, it is necessary to form meaningful configured port clusters that take the current traffic model into account.

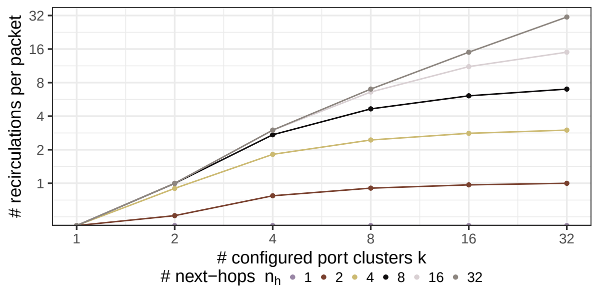
[5]This restores all bits from the original BIER bitstring that have not been processed yet.

VII. Port Clustering Algorithms for Efficient BIER Forwarding

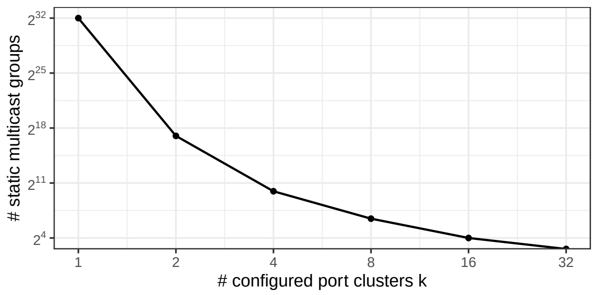
In this section, we first illustrate the optimization potential of efficient BIER forwarding through configuration of appropriate port clusters. Then, we present three clustering algorithms to reduce the average recirculations per packet: random port clustering (RPC) as a simple baseline, port clustering based on Spectral Clustering (PCSC), and recursive clustering with overlaps (RPCO) which also leverages Spectral Clustering for subroutines. For the latter two algorithms we present a graph embedding method that turns ports of sampled packets into a graph structure from which the algorithms learn correlated port clusters.

A. Optimization Potential and Approach

The bits in the BIER header require a packet to be sent to a certain set of next-hops, and, thereby, to specific ports



(a) Average number of recirculations per packet for $n_h \in \{1, 2, 4, 8, 16, 32\}$ random next-hops.



(b) Number of required static multicast groups.

Figure 6: Average number of recirculations and number of static multicast groups for $k \in \{1, 2, 4, 8, 16\}$ configured port clusters.

of a switch. To be brief, we talk about "ports of a packet". In the previous section we showed how multiple ports of a BIER packet can be served at once to speed up the forwarding process. For example, port clusters $\{1, .., 8\}$, $\{9, .., 16\}$, $\{17, .., 24\}$, and $\{25, .., 32\}$ may be configured. Then, a BIER packet needs to be processed at most four times, i.e., it must be recirculated at most three times, no matter how many BFERs are set in the BIER header. If a packet has only ports in the range $\{1, .., 8\}$, the packet does not need to be recirculated at all. However, if a packet has ports $\{1, 9, 17, 25\}$, it still requires three recirculations. The worst-case performance of the presented mechanism is therefore the number of configured port clusters $|\mathcal{C}|$ - 1 instead of #next-hops - 1. This also holds for the subsequently presented optimization algorithms RPC, PCSC, and RPCO.

We now assume that ports of a packet are not random but correlated. That is, certain ports sets tend to occur together. We call them correlated port clusters. We propose to learn these correlated port clusters from sampled traffic and to utilize them as configured port clusters. Then it is likely that BIER packets can be forwarded with fewer processing steps and, thereby, the number of recirculations may be reduced. In practice, a controller can continuously sample multicast traffic from a switch, learn the correlated port clusters of the sampled multicast traffic, and adjust the configured port clusters on the switch.

Large configured port clusters require lots of static multicast groups, but they have the potential to effectively reduce the number of recirculations. A constraint is the maximum number m_{max} of static multicast groups usable for configured port clusters which may be a technical limit or defined by the administrator.

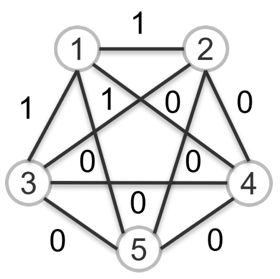
B. Random Port Clustering (RPC)

With RPC, n_p ports are randomly partitioned into approximately k equal-size clusters. The number of clusters k is determined such that the resulting number of configured static multicast groups is at most m_{max}. As the algorithm is trivial, we do not provide any further details. The method will serve as a baseline for a performance comparison.

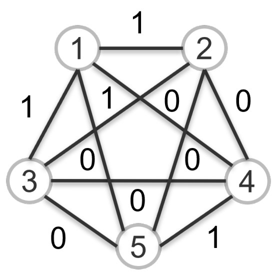
C. Port Clustering based on Spectral Clustering (PCSC)

We first present a graph embedding method that turns ports of sampled packets into a graph structure from which the algorithms learn correlated port clusters. Then we present the PCSC algorithm which is based on Spectral Clustering. It partitions n_p ports into approximately equal-size port clusters.

1) Graph Embedding: We embed the port information of sampled packets into a graph which is needed by the algorithms for PCSC and RPCO. The nodes of the graph represent the ports of a switch. The graph is fully connected and the edges have weights. All weights are initially zero. The embedding iteratively processes the sampled packets. For every combination of two ports of a packet, the weight of the link between these ports is increased by one. Figure 7(a)-Figure 7(b) illustrate how two sampled packets with ports $\{1, 2, 3\}$ and $\{4, 5\}$, respectively, modify an embedded graph with 5 nodes whose edges are initially all zero.



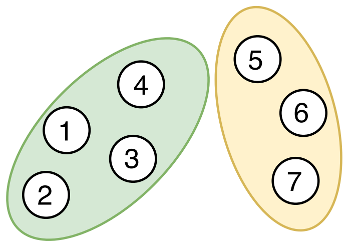
(a) The edge weights between egress ports 1, 2, and 3 are increased by one.



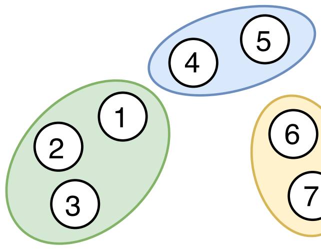
(b) The edge weights between egress ports 4 and 5 are increased by one.

Figure 7: Graph embedding: a full-mesh graph is augmented by port information from sampled packets: high edge weights indicate port pairs that frequently occur together in a BIER packet.

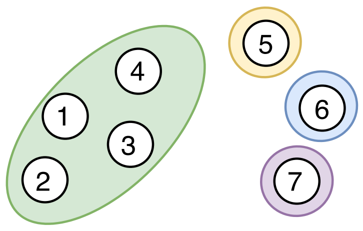
2) PCSC Algorithm: We first develop a metric for port clusters that correlates with the number of recirculations needed for the sampled traffic. Then we propose pseudocode for PCSC that minimizes that number while respecting the number of usable static multicast groups.



(a) Two almost equal-size port clusters need 15 static multicast groups.



(b) Three almost equal-size port clusters need 6 static multicast groups.



(c) The optimal port clusters have unequal size and need 11 static multicast groups.

Figure 8: Most BIER packets are sent to ports $\{1, 2, 3, 4\}$, $\{3, 4\}$, and $\{4, 5\}$ on a 7-port switch and the maximum number of usable static multicast groups is $m_{max} = 12$. PCSC produces equal-size port clusters while the optimum port clusters minimizing the overall number of recirculations has unequal size.

a) Metric: We consider two port clusters C_1 and C_2. The clustering is good if only a few BIER packets need to be sent through ports of C_1 and C_2. We identify a metric for the graph embedding that correlates with that number of packets although it is not an exact measure for it. The function $cut(C_1, C_2)$ is the sum of the weights on the edges between any two nodes $v_1 \in C_1$ and $v_2 \in C_2$. It gives an upper bound on the number of packets with ports in both C_1 and C_2. It is an upper bound and not the exact number as a packet may have multiple ports from C_1 and/or C_2. To assess whether the clustering is good, we need to relate $cut(C_1, C_2)$ to the overall number of nodes in the considered clusters. This can be done with the so-called normalized cut (Ncut) and is given below, generalized for multiple clusters.

$$Ncut(C_1, ..., C_k) = \sum_{i=1}^{k} \frac{cut(C_i, \overline{C_i})}{vol(C_i)}$$

Thereby, $cut(C_i, \overline{C_i})$ measures the sum of the edge weights between nodes in C_i and nodes that are not in C_i ($\overline{C_i}$). The function $vol(C_i)$ sums up the edge weights of all nodes in C_i – as a result, edge weights between nodes within the cluster are counted twice, weights of outgoing edges are counted once. The objective is to find clusters $C_1, ..., C_k$ that minimize the normalized cut. Ncut is known to be NP hard and therefore cannot be solved efficiently. However, Spectral Clustering is a relaxation of Ncut. It yields a partition $\mathcal{C}$ with preferably equal-size[6] clusters $C_i \in \mathcal{C}$ and can be solved efficiently [43].

b) Pseudocode for PCSC: PCSC is described in Algorithm 1. It first performs the graph embedding for the set of sampled packets $\mathcal{S}$ and the given number of nodes n_p. Then, Spectral Clustering is called to provide a partition $\mathcal{C}$ of the n_p nodes into k clusters. This is performed in a loop, starting from a single cluster up to n_p clusters. As soon as a partition $\mathcal{C}$ is found that requires at most m_{max} static multicast groups, the algorithm stops and $\mathcal{C}$ is returned. It is the clustering with the lowest number of clusters that can be configured with m_{max} static multicast groups.

[6]This property is desirable as the number of static multicast groups increases exponentially with the number of nodes in a cluster.

Algorithm 1 PCSC

```
Input: samples: $\mathcal{S}$
       number of ports: $n_p$
       number of multicast groups: $m_{max}$

$graph$ = $graphEmbedding(n_p, \mathcal{S})$
for $k$ from 1 to $n_p$ do
    $\mathcal{C}$ = SpectralClustering($graph$, $k$)
    if number of multicast groups for $\mathcal{C} \leq m_{max}$ then
        return $\mathcal{C}$
    end
end
```

D. Recursive Port Clustering with Overlap (RPCO)

We first explain two major shortcomings of PCSC. Then we explain how RPCO solves these shortcomings. Finally, we give a high-level pseudocode description of RPCO.

1) Shortcomings of PCSC: PCSC has two major shortcomings. First, if the configured port clusters cannot be built, the number of clusters is increased by one. As a result, an important cluster that significantly reduces the number of recirculations may not be built although a less important cluster could be split to save static multicast groups.

We illustrate that with a 7-port switch and $m_{max} = 12$ usable static multicast groups. We assume that most multicast packets are sent to port clusters $\{1, 2, 3, 4\}$, $\{3, 4\}$, and $\{4, 5\}$. When PCSC is called with $k = 2$, the clusters in Figure 8(a) may be returned which require 15 static multicast groups, which exceeds m_{max} so that it is not a valid solution. Therefore, PCSC increases k to 3, which may return the clusters in Figure 8(b) which require only 6 static multicast group. As this is feasible, this clustering is PCSC's final result. However, the optimal clustering that minimizes the overall number of recirculations might be the one in Figure 8(c) with 4 unequal-size clusters. They require 11 static multicast groups, which is also feasible.

Second, PCSC creates disjoint clusters. This, however, may not be optimal. We illustrate that by a small example. We consider packets with ports $\{1, 2, 3\}$ and $\{2, 3, 4\}$ and

$m_{max} = 8$ usable static multicast ports. A single, large cluster $C = \{1, 2, 3, 4\}$ requires $m(C) = 11$ static multicast groups so that it cannot be configured. When working with smaller, non-overlapping clusters, it is not possible to cover the port sets of both packets with only a single port cluster. However, when working with overlapping port clusters $C_1 = \{1, 2, 3\}$ and $C_2 = \{2, 3, 4\}$, only 7 static multicast groups are needed[7], which is feasible. Moreover, the port sets of both packets can be covered. Thus, overlapping clusters may help to further reduce the number of recirculations with a limited number of usable static multicast groups.

2) Design Ideas: We discuss major design ideas of RPCO. If the number of usable static multicast groups m_{max} does not suffice to configure k clusters proposed by Spectral Clustering, RPCO selects the clusters that reduce recirculations in the most efficient way and recursively re-clusters the remaining clusters. To that end, we review and adapt the knapsack algorithm to select the clusters that reduce recirculations most efficiently. Given a clustering, we further suggest how to add nodes also to other clusters they are not yet part of, which facilitates cluster overlaps.

a) The Knapsack Algorithm: In the knapsack problem [50], a set of items is given, and each item has a weight and a value. The knapsack objective is to select items such that their overall weight is less than a given limit while their overall value is maximized.

We apply the knapsack algorithm as follows. The set of items is given as set of port clusters $\mathcal{C} = \{C_1, ..., C_k\}$. The value of a port cluster C_i is given by the number of recirculation it saves for the set of packets $\mathcal{S}$ which is evaluated by simulation. The weight of a port cluster C_i is given by its number of required static multicast groups $m(C_i)$. The limit is the number of usable static multicast groups. The algorithm selects those clusters that maximize the number of saved recirculations with the available static multicast groups.

b) Adding Single Nodes to Multiple Clusters: We first define the so-called port-cluster relevance $r(x, C)$ of a port x and a cluster C, $x \notin C$. Then, we explain how the port-cluster relevance is used to add single nodes to multiple clusters.

The port-cluster relevance measures the connectivity between port x and cluster C. It is the sum of the edge weights w between x and C, i.e., $r(x, C) = \sum_{y \in C} w(x, y)$.

Ports are initially assigned to a cluster with Spectral Clustering. However, ports may also be important for other clusters. The list of all port-cluster pairs sorted by decreasing port-cluster relevance suggests the order in which nodes should be additionally added to another cluster provided the remaining static multicast groups suffice. As a result, a partition of ports becomes a port clustering with overlaps.

3) Pseudocode for RPCO: We give a high-level pseudocode for RPCO and refer to the Github repository[8] for details.

Algorithm 2 describes the outer control loop of RPCO. First, the graph embedding of the samples $\mathcal{S}$ is computed and stored in $graph$. Then, the best clustering $\mathcal{C}_{best}$ is initialized with single node clusters. A graph with n_p nodes (number of ports on the switch) can be partitioned into up to n_p clusters. Therefore, the subsequent loop is called with k between 1 and n_p. Within the loop, the current clustering $\mathcal{C}$ is initialized empty and the number of remaining static multicast groups m_{left} is initialized with m_{max}. Both $\mathcal{C}$ and m_{left} are global variables so that they can be modified by subroutines. RecursiveClustering computes a partition of all nodes and stores it in $\mathcal{C}$. Details of the procedure will be explained later. Then, OverlapClusters utilizes remaining usable static multicast groups m_{left} to add nodes to other clusters they are not yet part of (see Section VII-D2b). This leads to overlapping clusters. Afterwards, the best clustering $\mathcal{C}_{best}$ is updated by $\mathcal{C}$ if $\mathcal{C}$ requires fewer recirculations than the best clustering. The function Recirculations$(\mathcal{C}, \mathcal{S})$ computes the number of recirculations required for clustering $\mathcal{C}$ for the packets in $\mathcal{S}$. Finally, RPCO returns the best clustering of all switch ports that minimizes the number of recirculations for the samples $\mathcal{S}$.

Algorithm 2 RPCO

```
Input: samples: $\mathcal{S}$
       number of ports: $n_p$
       max. number of multicast groups: $m_{max}$
$graph$ = GraphEmbedding$(n_p, \mathcal{S})$
$\mathcal{C}_{best} = \{\{1\}, ...\{n_p\}\}$
for $k \in [1, n_p]$ do
    $\mathcal{C} = \{\emptyset\}$
    $m_{left} = m_{max}$
    RecursiveClustering$(graph, k)$
    OverlapClusters$(graph)$
    if Recirculations$(\mathcal{C}, \mathcal{S})$ < Recirculations$(\mathcal{C}_{best}, \mathcal{S})$ then
        $\mathcal{C}_{best} = \mathcal{C}$
    end
end
return Best port clustering $\mathcal{C}_{best}$
```

RecursiveClustering is described in Algorithm 3. If the graph contains only a single node v, the node is added as a separate cluster to $\mathcal{C}$ and the recursion ends. Otherwise, Spectral Clustering is executed to produce clustering $\mathcal{C}'$ with the desired number of clusters k. Then, the cluster set $\mathcal{C}^*$ is identified which makes best use of the remaining static multicast groups m_{left} to reduce recirculations. All clusters in $\mathcal{C}^*$ are added to the current clustering result $\mathcal{C}$ and m_{left} is decreased by their number of required static multicast groups. The clusters not selected by knapsack ($\mathcal{C}' \setminus \mathcal{C}^*$) are recursively clustered. To that end, the corresponding embedded subgraph is computed. The recursion ends if either the recursion was called with a single node or if all clusters $\mathcal{C}'$ can be selected.

VIII. Simulative Performance Comparison

In this section we compare the performance of the three port clustering methods Random Port Clustering (RPC), Port Clustering based on Spectral Clustering (PCSC), and Recursive Port Clustering with Overlaps (RPCO). We first develop a model for correlated multicast traffic and explain the

[7]When working with overlapping port clusters, the static multicast groups required by multiple port clusters need to be configured only once on the switch.

[8]Github: https://github.com/uni-tue-kn/rpco

Algorithm 3 RecursiveClustering

```
Input: graph embedding: graph
       number of clusters: $k$

if $graph$ contains only the single node $v$ then
    $\mathcal{C} = \mathcal{C} \cup \{\{v\}\}$
    return
end
$\mathcal{C}'$ = SpectralClustering($graph, k$)
$\mathcal{C}^* =$ knapsack($\mathcal{C}', \mathcal{S}, m_{left}$)
for $C \in \mathcal{C}^*$ do
    $\mathcal{C} = \mathcal{C} \cup \{C\}$
    $m_{left} = m_{left} - m(C)$
end
for $C \in \mathcal{C}' \setminus \mathcal{C}^*$ do
    $subgraph$ = subgraph of $graph$ limited to nodes in $C$
    RecursiveClustering($subgraph, 2$)
end
```

performance evaluation methodology. Then, we compare the performance of the mentioned clustering methods for various correlated multicast traffic models. Finally, we compare the runtime of the algorithms and discuss their scalability properties.

A. Traffic Model and Evaluation Methodology

We define a simple model for correlated multicast traffic and explain the methodology for the subsequent comparison of the port clustering methods.

1) Model for Correlated Multicast Traffic: In Section VI-C we utilized a model for multicast traffic that assumes random ports for subsequent multicast packets. However, random ports are not realistic for two reasons. First, subsequent multicast packets belong to a set of active multicast groups and packets of a multicast group have identical ports as long as the groups do not change. Second, receivers of multicast groups are users or connected upstream aggregation points in specific time zones, geographical regions, or neighborhoods. Therefore, we assume the users have common interests for certain multicast content so that they belong to multicast groups with correlated receivers. We have not found any literature studying this issue and think this would be useful future work.

We propose a model for correlated multicast traffic for use in the subsequent performance comparison. We define a set of generating port clusters $\mathcal{C}_g = \{C_1, C_2, ..., C_k\}$ from which ports of a packet are preferentially chosen. First, we randomly choose one generating port cluster C_i; thereby all C_i have equal probability. Then, we determine a random number of ports which is equally distributed between 1 and the size $|C_i|$ of the chosen cluster. We draw these ports with a probability p from C_i (without duplicates) and with probability $1-p$ from ports outside C_i (without duplicates).

For $p = 1$, all ports of a sampled BIER packet are from a single, generating port cluster C_i. In that case, if the generating port clusters $\mathcal{C}_g$ are configured for efficient BIER forwarding, BIER packets can be forwarded without recirculation. As p decreases, a sampled BIER packet is likely to have increasingly more ports outside the selected generating port cluster C_i. That means, the resulting multicast traffic is more random and more recirculations are needed. We take p as a measure for port correlation in the generated multicast traffic.

2) Evaluation Methodology: The objective of port clustering algorithms for efficient BIER forwarding is the reduction of recirculations. Therefore, we take the average number of recirculations per packet as performance metric for the subsequent comparisons.

We generate 1000 BIER packets. Based on these packets we compute sets of port clusters for optimized configuration using the considered port clustering methods and various numbers of usable static multicast ports m_{max}. Then, we generate another 10000 packets and simulate efficient BIER forwarding using the optimized configuration. We count the number of recirculations and compute the average number of recirculations per packet. We conduct the experiments 100 times and produce 95% confidence intervals for the average number of recirculations. As they are very small, we omit them in the figures for the sake of readability.

B. Performance Comparison of Port Clustering Methods

We compare the efficiency of the port clustering algorithms for different traffic models. We consider disjoint and overlapping generating port clusters of equal and unequal size. We choose the models such that they all lead to 4.5 ports per BIER packet, which makes their results comparable.

1) Multicast Traffic Generated from Disjoint Port Clusters: We study correlated multicast traffic generated from disjoint generating port clusters. We consider symmetric and asymmetric generating port clusters.

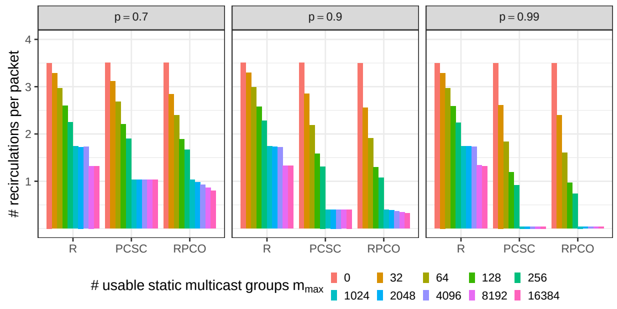
a) Symmetric Generating Port Clusters: We consider four symmetric, disjoint, generating port clusters of size 8: $C_1 = \{1, .., 8\}$, $C_2 = \{9, .., 16\}$, $C_3 = \{17, .., 24\}$, $C_4 = \{25, .., 32\}$. If they are used for configuration, $4 \cdot (2^8 - 8 - 1) = 988$ static multicast groups are needed.

Figure 9(a) shows the average number of recirculations per packet for traffic models with port correlation $p \in \{0.7, 0.9, 0.99\}$, for usable static multicast groups $m_{max} \in \{0, 32, 64, 128, 256, 10.24, 2048, 4096, 8192, 16384\}$, and for the port clustering methods RPC, PCSC, and RPCO.

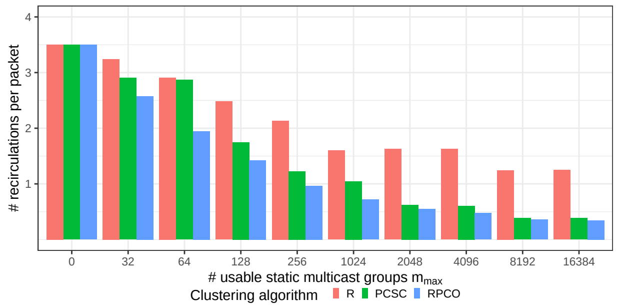
If no static multicast group is available for efficient BIER forwarding ($m_{max} = 0$), the port clustering is disabled, and the forwarding behaviour is the same as the one for simple BIER forwarding. Therefore, packets with 4.5 ports on average require 3.5 recirculations on average. Increasing the number of usable multicast groups m_{max} allows efficient BIER forwarding to decrease the average number of recirculations per packet. This holds for all traffic models and for all port clustering methods. However, if sufficient static multicast groups are available, the degree to which the average number of recirculations can be reduced depends on the port correlation p and the port clustering method.

If a packet with l ports is generated from a specific generating port cluster, all the ports are taken from that cluster with a probability of p^l. Setting $l = 4.5$ yields 20.1% for $p = 0.7$, 62.2% for $p = 0.9$, and 95.6% for $p = 0.99$.

Thus, the chosen traffic models are quite divers. For port correlation $p = 0.7$, the average number of recirculations are similar for all considered port clustering algorithms. The advanced port clustering algorithms hardly outperform the random method due to the lack of sufficient port correlation in the generated multicast traffic. For port correlation $p = 0.99$, most packets are entirely drawn from a single generating port cluster. As a result, the advanced packet clustering methods lead to significantly fewer packet recirculations than random clustering. With $m_{max} = 1024$ or more usable multicast groups, PCSC and RPCO reduce the average number of recirculations to almost zero. Apparently they are able to learn the right port clusters. The generating port clusters are optimal for configuration; as mentioned above, they require 988 static multicast groups. This explains why $m_{max} = 512$ or fewer static multicast groups require more recirculations, also with advanced port clustering methods. The results in Figure 9(a) show that PSCS and RPCO lead to about the same number of recirculations per packet for symmetric, disjoint, generating port clusters.



(a) Traffic sampled from four generating port clusters of size 8 with different port correlation p.

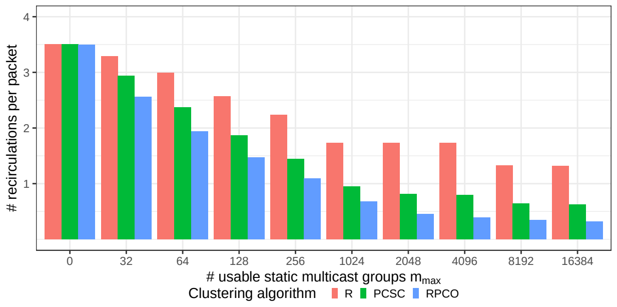


(b) Traffic sampled from four clusters of size 12, 10, 6, 4 with port correlation $p = 0.9$.

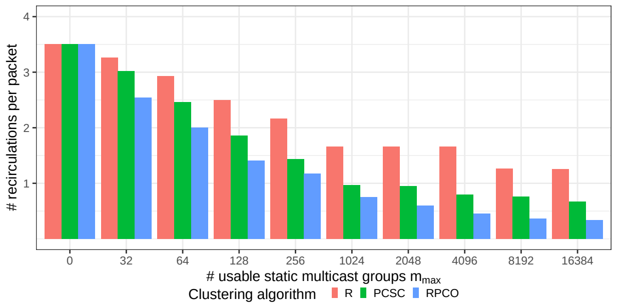
Figure 9: Impact of port clustering methods and number m_{max} of usable, static multicast groups on the average number of recirculations per packet; multicast traffic is sampled from disjoint generating port clusters.

In the following, we choose port correlation $p = 0.9$ as this generates sufficiently correlated multicast traffic with substantial port deviation from the generating port clusters.

b) Asymmetric Generating Port Clusters: We consider four asymmetric, disjoint, generating port clusters of size 12, 10, 6, 4: $C_1 = \{1, .., 12\}$, $C_2 = \{13, .., 22\}$, $C_3 = \{23, .., 28\}$, and $C_4 = \{29, .., 32\}$. If used for configuration, they require $(2^{12}-12-1)+(2^{10}-10-1)+(2^6-6-1)+(2^4-4-1) = 5164$ static multicast groups.



(a) Traffic sampled from six clusters of size 8.



(b) Traffic sampled from six clusters of size 12, 10, 8, 8, 6, 4.

Figure 10: Impact of port clustering methods and number m_{max} of usable, static multicast groups on the average number of recirculations per packet; multicast traffic is sampled from overlapping, generating port clusters with port correlation $p = 0.9$.

Figure 9(b) illustrates the average number of recirculations per packet for port correlation $p = 0.9$. Again, more usable static multicast groups cause fewer recirculations. We now observe that RPCO reduces the average number of recirculations to lower numbers than PCSC, in particular for $m_{max} \leq 4096$. For larger m_{max}, PCSC and RPCO lead to almost equal results. This is in line with the design goal of RPCO: it makes better use of a limited number of static multicast groups than PCSC by proposing unequal-size port clusters. For $m_{max} = 64$, PCSC causes 3 recirculations per packet while RPCO causes only 2. For port correlation $p = 0.99$, which is not shown in the figure, both PCSC and RPCO reduce the average number of recirculations to almost zero for $m_{max} \geq 8192$.

2) Multicast Traffic Generated from Overlapping Port Clusters: We study the performance of the presented clustering algorithms for overlapping, generating port clusters.

a) Symmetric Generating Port Clusters: We consider six overlapping, generating port clusters of size 8: $C_1 = \{1, .., 8\}$, $C_2 = \{6, .., 13\}$, $C_3 = \{11, .., 18\}$, $C_4 = \{17, .., 24\}$, $C_5 = \{22, .., 29\}$, and $C_6 = \{28, .., 32, 1, .., 3\}$. Configuring them as port clusters requires $6 \cdot (2^8 - 8 - 1) - 4 \cdot (2^3 - 3 - 1) - 2 \cdot (2^2 - 2 - 1) = 1464$ static multicast groups.

Figure 10(a) indicates the average number of recirculations per packet for port correlation $p = 0.9$. Here, PCSC outperforms RPC, and RPCO outperforms PCSC for any number $m_{max} > 0$ of usable static multicast groups. While PCSC computes only disjoint port clusters, RPCO may yield overlapping port clusters. This can lead to fewer recirculations when frequently observed port groups of packets are partly overlapping. For port correlation $p = 0.99$, which is not shown in the figure, only RPCO reduces the average number of recirculations to almost zero for $m_{max} \geq 2048$.

b) Asymmetric Generating Port Clusters: We consider six overlapping, generating port clusters of size 12, 10, 8, 8, 6, 4: $C_1 = \{1, .., 12\}$, $C_2 = \{27, .., 32, 1, .., 4\}$, $C_3 = \{9, .., 16\}$, $C_4 = \{22, .., 29\}$, $C_5 = \{18, .., 23\}$, and $C_6 = \{16, .., 19\}$. Configuring them as port clusters requires 5630 static multicast groups.

Figure 10(b) illustrates the average number of recirculations per packet for port correlation $p = 0.9$. The results are very similar to those in Figure 10(a), only a few recirculations more are required. That means, PCSC clearly outpeforms RPC, and RPCO outperforms PCSC. For $p = 0.99$ and $m_{max} \geq 8192$, which is not shown here, RPCO even reduces the average number of recirculations to almost zero. That is, it is able to find optimal clusters for configuration even under challenging conditions (overlapping, unequal-size, generating port clusters).

C. Runtime

The presented clustering algorithms, especially RPCO, seem rather complex at first glance. We measure the runtime of the presented algorithms for the evaluation in Section VIII-B2b. The experiments are executed on a 2022 Mac Studio with M1 Max and 32 GB of RAM. Figure 11 compiles the results.

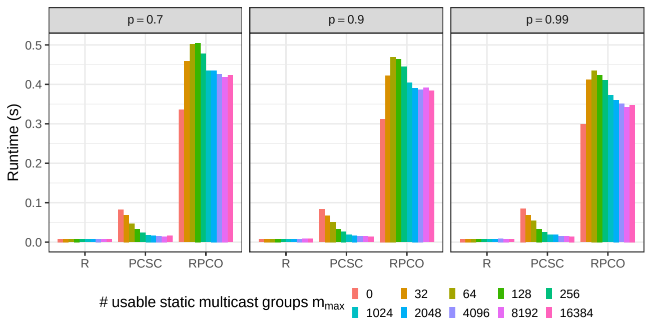


Figure 11: Average runtime in seconds for the clustering algorithms RPC, PCSC, and RPCO while performing experiments for Section VIII-B2b.

Random Port Clustering (RPC) has the shortest runtime with at most 9 ms. It partitions all ports into equal-size clusters and its runtime is therefore independent of port correlation p. PCSC reveals the second lowest runtime with up to 86 ms. It calls the Spectral Clustering subroutine at most n_p times where n_p is the number of ports. PCSC's runtime decreases with increasing m_{max} because larger values of m_{max} lead to fewer subroutine calls (return leaves the loop in Algorithm 1). RPCO has the longest runtime with up to 527 ms. It also performs n_p iteration steps but may call Spectral Clustering multiple times within a single iteration step. Its runtime primarily depends of the number of recursive calls. With decreasing p, RPCO's runtime decreases. Lower values of p lead to more uncorrelated packets, which leads to a blurred graph structure in the sense of more homogeneous edge weights. The Spectral Clustering subroutine tends to return larger clusters on a blurred graph. When not all clusters can be built, RPCO recursively re-clusters them. This is more likely with a blurred graph structure than with a sharp graph structure, i.e., a higher correlation between packets.

Although RPCO has the longest runtime, RPCO can be carried out sufficiently fast so that it can be well applied in practice as configured port clusters may be adapted rather on the time scale of minutes than seconds.

D. Scalability

In the following, we discuss the scalability properties of the presented mechanisms, i.e., how they behave in larger networks. First, the presented clustering algorithms leverage only local information for optimization, i.e., they only require sampled packets from a switch. Therefore, the optimization is preferably done by a controller running on the switch itself, which eliminates the need for additional control plane traffic in the network. Second, the used graph embedding (Section VII-C1) has a constant size per switch, i.e., it scales linearly with the number of switch ports. Therefore, the runtime is bounded by a small constant for a realistic number of maximal ports of a switch. As a consequence, the presented mechanisms are highly scalable and also suited for large networks.

IX. Experimental Performance Evaluation

In this section we perform experiments in a hardware testbed to demonstrate the practical feasibility of the proposed concepts and to validate the theoretical results from Section VIII. First, we explain the concept and the testbed setup. Then, we describe the performed experiments.

A. Concept

Figure 12 illustrates the concept for the hardware testbed.

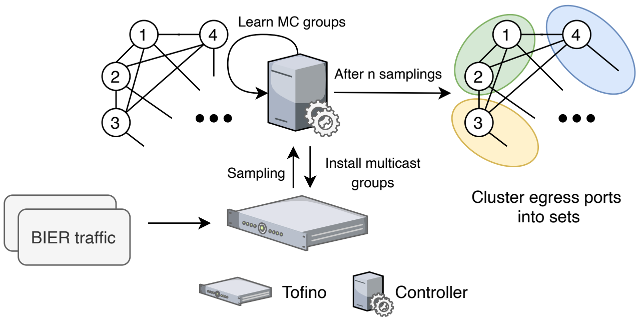


Figure 12: Concept for the hardware evaluation.

The Tofino [48], a P4-programmable switching ASIC, is the core of the hardware testbed. We utilize a Tofino-based Edgecore Wedge 100BF-32X switch [49] with 32 100 Gbit/s ports that runs the adapted BIER implementation as described in Section VI. BIER traffic is sampled at the Tofino with a rate of 0.1%, i.e., every 1000^{th} BIER packet. Sampled packets are sent to the controller and used for the graph embedding as described in Section VII. For 100 Gbit/s incoming multicast traffic, this amounts to 100 Mbit/s which can be efficiently handled by the controller. Alternatively, the number of sampled packets can also be limited through a Meter[9] instance. After 2^{10} samples, the controller applies the optimization heuristic and installs the static multicast groups of the configured port clusters. We measure the average recirculation traffic on the Tofino to assess the effectiveness of the presented optimization heuristics. To that end, packets on the recirculation port are cloned to a separate end host that measures the incoming bandwidth which equals the rate of the recirculation traffic.

B. Traffic Generation

Generating UDP traffic at high rate according to a given distribtion is a difficult task. We leverage Iperf [52] to generate homogeneous UDP traffic on an end host. It is sent to the Tofino which adapts it according to a specified distribution of BIER headers. When the Tofino receives a UDP packet generated by Iperf, it generates a random number between 0 and $2^w - 1$, where w is a parameter of the random extern on Tofino that generates a random number between 0 and $2^w - 1$. The generated random number is then used as index to a match-action table that maps the random number to a BIER header (see Figure 13). Then, the header of the UDP packet is substituted by the BIER header indicated in the table. Thereby, a UDP packet stream with any distribution of BIER headers can be generated.

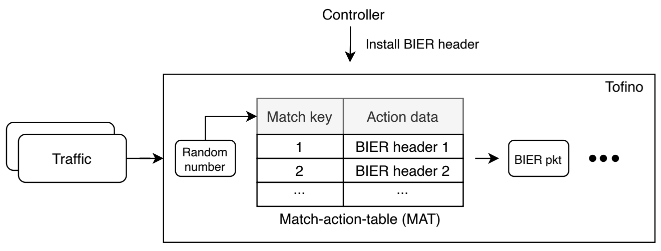


Figure 13: A match-action table is used to turn homogeneous UDP traffic into BIER traffic with headers following a desired distribution.

The match-action tables is populated a priori by a controller which has sampled 2^w BIER headers according to the traffic model in Section VIII-A1 for a given set of generating port clusters and a port correlation p. As a result, the Tofino turns homogeneous UDP traffic into BIER traffic whose headers follow a desired distribution.

[9]Intel Tofino supports 3-color metering as described in [51].

C. Experiment

We validate our hardware implementation by conducting the same experiments as in Section VIII-B2b. Thus, the traffic model consists of six overlapping generating port clusters of size 12, 10, 8, 8, 6, and 4. We choose port correlation $p = 0.9$, and use $w = 14$ to install 2^w sampled BIER headers of that distribution in the match-action table on the Tofino. We generate 5 Gbit/s UDP traffic via Iperf and send it to the Tofino which turns it into BIER traffic with the desired header distribution. We perform 5 runs per experiment and report average values.

The controller samples the BIER traffic and computes optimized port clusters for configuration on the Tofino. Thereby, different port clustering methods and different numbers m_{max} of usable static multicast groups are considered. Figure 14 shows the average recirculation traffic in Gbit/s.

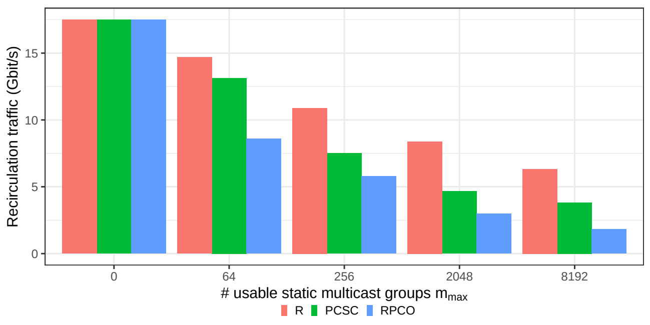


Figure 14: Average recirculation traffic for RPC, PCSC and RPCO and different numbers m_{max} of usable static multicast groups; the traffic model has six overlapping generating port clusters and port correlation $p = 0.9$; the results are to be compared with those in Figure 10(b).

If no static multicast group is available ($m_{max} = 0$), efficient BIER forwarding is essentially disabled, and the observed behaviour is the same as the one for simple BIER forwarding. Therefore, packets with 4.5 ports on average require 3.5 recirculations on average, which results in $3.5 \cdot 5$ Gbit/s $= 17.5$ Gbit/s recirculation traffic. This closely matches the results of Section VIII-B2b. An increasing number m_{max} of usable static multicast groups decreases the average number of recirculations per packet and therefore the recirculation traffic. Again, PCSC and RPCO clearly outperform RPC and RPCO performs better than PCSC (for $m_{max} > 0$). In fact, for $m_{max} = 8192$, RPCO reduces the recirculation traffic by 71% compared to RPC and 52% compared to PCSC. The experimental results in Figure 14 are in line with the simulation results in Figure 10(b) as they show the same proportions.

We performed this experiment with only 5 Gbit/s incoming traffic due to the lack of a fast generator for contant bit rate traffic. However, efficient BIER forwarding runs at line rate at the Tofino[10], i.e., it is capable of handling 32×100 Gbit/s incoming traffic.

[10]Every P4 program that compiles for the Tofino runs at line rate.

X. Conclusion

Bit Index Explicit Replication (BIER) forwards multicast traffic without signalling and states within BIER domains. Thereby, it greatly improves scalability for multicast in core networks. However, a simple implementation of that concept implies iterative packet transmission which requires additional processing capatity [2] on a single switch. In this paper we presented efficient BIER forwarding with static multicast groups such that a BIER packet can be sent to multiple next-hops in a single pipeline iteration. To that end, we configure port clusters on the switch and install all combinations of ports within each port cluster as static multicast group. Simple match-action operations choose the appropriate port clusters and therein the right static multicast group so that packets are transmitted to multiple next-hops in a single iteration step. As a result, a BIER packet can be processed in high-speed with a single or at most a few iteration steps. We demonstrated by simulation that randomly selected disjoint equal-size configured port clusters can decrease the required recirculations by 90% with only 1024 static multicast groups on a 32 port switch with 32 next-hops (Section VI-C) compared to simple iterative BIER forwarding. Further, we presented port clustering algorithms based on Spectral Clustering which learn the current BIER traffic pattern and compute port clusters for configuration. Recursive Port Clustering with Overlap (RPCO) reduces the required recirculations by up to 96% compared to randomly selected port clusters (Section VIII). We implemented efficient BIER forwarding on the Edgecore Wedge 100BF-32X, a 32 100 Gbit/s port high-performance P4 switch, and validated the simulation results in a hardware testbed.

The work comes with a few byproducts. We developed efficient BIER forwarding for data plane programming with the Tofino ASIC. Other switch architectures will also face the challenge to determine outgoing ports of a BIER packet with little effort and can benefit from the presented algorithms. We proposed a traffic model for the outgoing ports of multicast traffic on a switch for evaluation purposes. Future work may validate that traffic model based on measured data. Finally, we developed a simple method for data plane programming to modify traffic such that its headers correspond to a specific distribution. This may also be useful in other experimental work.

References

[1] I. Wijnands *et al.*, *RFC 8279: Multicast Using Bit Index Explicit Replication (BIER)*, https://datatracker.ietf.org/doc/rfc8279/, Nov. 2017.

[2] D. Merling *et al.*, “Hardware-Based Evaluation of Scalable and Resilient Multicast With BIER in P4,” *IEEE Access*, vol. 9, pp. 34 500–34 514, 2021.

[3] S. Islam *et al.*, “A Survey on Multicasting in Software-Defined Networking,” *IEEE Communications Surveys Tutorials (COMST)*, vol. 20, pp. 355–387, 2018.

[4] Z. Al-Saeed *et al.*, “Multicasting in Software Defined Networks: A Comprehensive Survey,” *Journal of Network and Computer Applications (JNCA)*, vol. 104, pp. 61–77, 2018.

[5] M. Shahbaz *et al.*, “Elmo: Source Routed Multicast for Public Clouds,” in *ACM SIGCOMM*, 2019, pp. 458–471.

[6] A. Iyer *et al.*, “Avalanche: Data Center Multicast using Software Defined Networking,” in *International Conference on Communication Systems and Networks*, 2014, pp. 1–8.

[7] W. Cui *et al.*, “Scalable and Load-Balanced Data Center Multicast,” in *IEEE GLOBECOM*, 2015, pp. 1–6.

[8] X. Li *et al.*, “Scaling IP Multicast on Datacenter Topologies,” in *ACM CoNEXT*, 2013, pp. 61–72.

[9] X. Zhang *et al.*, “A Centralized Optimization Solution for Application Layer Multicast Tree,” *IEEE Transactions on Network and Service Management (TNSM)*, vol. 14, pp. 771–785, 2017.

[10] K. Mokhtarian *et al.*, “Minimum-delay multicast algorithms for mesh overlays,” *IEEE/ACM Transactions on Networking*, vol. 23, pp. 973–986, 2015.

[11] S.-H. Shen, “Efficient SVC Multicast Streaming for Video Conferencing With SDN Control,” *IEEE Transactions on Network and Service Management (TNSM)*, vol. 16, pp. 403–416, 2019.

[12] M. A. Kaafar *et al.*, “A Locating-First Approach for Scalable Overlay Multicast,” in *IEEE INFOCOM*, 2006, pp. 1–2.

[13] R. Boivie, N. Feldman, and C. Metz, “Small Group Multicast: A New Solution for Multicasting on the Internet,” *IEEE Internet Computing*, vol. 4, pp. 75–79, 2000.

[14] A. Boudani and B. Cousin, “SEM: A New Small Group Multicast Routing Protocol,” in *International Conference on Telecommunications (ICT)*, 2003, pp. 450–455.

[15] W. K. Jia *et al.*, “A Unified Unicast and Multicast Routing and Forwarding Algorithm for Software-Defined Datacenter Networks,” *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 31, pp. 2646–2657, 2013.

[16] C. A. S. Oliveira *et al.*, “Steiner Trees and Multicast,” *Mathematical Aspects of Network Routing Optimization*, vol. 53, pp. 29–45, 2011.

[17] L. H. Huang *et al.*, “Scalable and Bandwidth-Efficient Multicast for Software-Defined Networks,” in *IEEE GLOBECOM*, 2014, pp. 1890–1896.

[18] J.-R. Jiang *et al.*, “Constructing Multiple Steiner Trees for Software-Defined Networking Multicast,” in *Conference on Future Internet Technologies*, 2016, pp. 1–6.

[19] Z. Hu *et al.*, “Multicast Routing with Uncertain Sources in Software-Defined Network,” in *IEEE/ACM International Symposium on Quality of Service (IWQoS)*, 2016, pp. 1–6.

[20] S. Zhou *et al.*, “Cost-Efficient and Scalable Multicast Tree in Software Defined Networking,” in *Algorithms and Architectures for Parallel Processing*, 2015, pp. 592–605.

[21] S.-H. Shen *et al.*, “Reliable Multicast Routing for Software-Defined Networks,” in *IEEE INFOCOM*, 2015, pp. 181–189.

[22] B. Ren *et al.*, “The Packing Problem of Uncertain Multicasts,” *Concurrency and Computation: Practice and Experience*, vol. 29, 2017.

[23] M. Popovic *et al.*, “Performance Comparison of Node-Redundant Multicast Distribution Trees in SDN Networks,” *International Conference on Networked Systems*, pp. 1–8, 2017.

[24] D. Kotani *et al.*, “A Multicast Tree Management Method Supporting Fast Failure Recovery and Dynamic Group Membership Changes in OpenFlow Networks,” *Journal of Information Processing (JIP)*, vol. 24, pp. 395–406, 2016.

[25] T. Pfeiffenberger *et al.*, “Reliable and Flexible Communications for Power Systems: Fault-tolerant Multicast with SDN/OpenFlow,” in *IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2015, pp. 1–6.

[26] J. Rückert *et al.*, “Software-Defined Multicast for Over-the-Top and Overlay-based Live Streaming in ISP Networks,” *Journal of Network and Systems Management (JNSM)*, vol. 23, pp. 280–308, 2015.

[27] J. Rueckert *et al.*, “Flexible, Efficient, and Scalable Software-Defined Over-the-Top Multicast for ISP Environments With

DynSdm," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 13, pp. 754–767, 2016.

[28] T. Humernbrum *et al.*, "Towards Efficient Multicast Communication in Software-Defined Networks," in *IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2016, pp. 106–113.

[29] Y.-D. Lin *et al.*, "Scalable Multicasting with Multiple Shared Trees in Software Defined Networking," *Journal of Network and Computer Applications (JNCA)*, vol. 78, pp. 125–133, 2017.

[30] M. J. Reed *et al.*, "Stateless Multicast Switching in Software Defined Networks," in *IEEE International Conference on Communications (ICC)*, 2016, pp. 1–7.

[31] W. Braun *et al.*, "Demo: Scalable and Reliable Software-Defined Multicast with BIER and P4," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2017, pp. 905–906.

[32] D. Merling *et al.*, "P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast," *Journal of Network and Computer Applications (JNCA)*, vol. 169, 2020.

[33] p4lang, *Behavioral-model*, https://github.com/p4lang/behavioral-model, Accessed: 2021-01-28, 2021.

[34] A. Bas, *BMv2 Throughput*, https://github.com/p4lang/behavioral-model/issues/537\#issuecomment-360537441, Jan. 2018.

[35] A. Giorgetti *et al.*, "First Demonstration of SDN-based Bit Index Explicit Replication (BIER) Multicasting," in *IEEE European Conference on Networks and Communications (EuCNC)*, 2017, pp. 1–6.

[36] A. Giorgetti *et al.*, "Bit Index Explicit Replication (BIER) Multicasting in Transport Networks," in *International Conference on Optical Network Design and Modeling (ONDM)*, 2017, pp. 1–5.

[37] Y. Desmouceaux *et al.*, "Reliable Multicast with B.I.E.R.," *Journal of Communications and Networks*, vol. 20, pp. 182–197, 2018.

[38] T. Eckert *et al.*, *Traffic Engineering for Bit Index Explicit Replication BIER-TE*, http://tools.ietf.org/html/draft-eckert-bier-te-arch, Nov. 2017.

[39] T. Eckert and B. Xu, *Carrier Grade Minimalist Multicast (CGM2) using Bit Index Explicit Replication (BIER) with Recursive BitString Structure (RBS) Addresses*, https://datatracker.ietf.org/doc/html/draft-eckert-bier-cgm2-rbs-01, Feb. 2022.

[40] W. Braun *et al.*, "Performance Comparison of Resilience Mechanisms for Stateless Multicast using BIER," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2017, pp. 230–238.

[41] J. B. MacQueen, "Some Methods for Classification and Analysis of MultiVariate Observations," in *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, University of California Press, 1967, pp. 281–297.

[42] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 226–231.

[43] U. v. Luxburg, "A Tutorial on Spectral Clustering," *Statistics and Computing*, vol. 17, pp. 395–416, 2007.

[44] H. Chen, M. McBride, S. Lindner, *et al.*, "BIER Fast ReRoute," Internet Engineering Task Force, Internet-Draft draft-chen-bier-frr-04, Jan. 2022, Work in Progress, 31 pp. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-chen-bier-frr-04.

[45] D. Merling, S. Lindner, and M. Menth, "Comparison of Fast-Reroute Mechanisms for BIER-Based IP Multicast," in *2020 Seventh International Conference on Software Defined Systems (SDS)*, 2020, pp. 51–58.

[46] P. Bosshart *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 87–95, 2014.

[47] F. Hauser, M. Haeberle, D. Merling, *et al.*, "A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research," *Accepted at Journal of Network and Computer Applications (JNCA)*, 2022.

[48] Intel, *Intel Tofino*, https://www.intel.de/content/www/de/de/products/network-io/programmable-ethernet-switch/tofino-series.html, Accessed: 2022-03-30, 2021.

[49] Edge-Core Networks, *Wedge100BF-32X/65X Switch*, https://www.edge-core.com/_upload/images/Wedge_100-32X_DS_R04_20170615.pdf, Accessed: 2022-12-28, 2021.

[50] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[51] J. Heinanen and R. Guerin, *RFC2698: A Two Rate Three Color Marker*, https://www.rfc-editor.org/info/rfc2698, Sep. 1999.

[52] iperf2 team, *iperf*, https://iperf.fr, Accessed: 2022-03-30.

Steffen Lindner is a Ph.D. student at the chair of communication networks of Prof. Dr. habil. Michael Menth at the Eberhard Karls University Tuebingen, Germany. He obtained his master's degree in 2019 and afterwards, became part of the communication networks research group. His research interests include software-defined networking, P4 and congestion management.

Daniel Merling is a Ph. D. student at the chair of communication networks of Prof. Dr. habil. Michael Menth at the Eberhard Karls University Tuebingen, Germany. There he obtained his master's degree in 2017 and afterwards, became part of the communication networks research group. His area of expertise include software-defined networking, scalability, P4, routing and resilience issues, multicast and congestion management.

Michael Menth, (Senior Member, IEEE) is professor at the Department of Computer Science at the University of Tuebingen/Germany and chairholder of Communication Networks since 2010. He studied, worked, and obtained diploma (1998), PhD (2004), and habilitation (2010) degrees at the universities of Austin/Texas, Ulm/Germany, and Wuerzburg/Germany. His special interests are performance analysis and optimization of communication networks, resilience and routing issues, as well as resource and congestion management. His recent research focus is on network softwarization, in particular P4-based data plane programming, Time-Sensitive Networking (TSN), Internet of Things, and Internet protocols. Dr. Menth contributes to standardization bodies, notably to the IETF.

1.5 P4TG: 1 Tb/s Traffic Generation for Ethernet/IP Networks

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2022.0092316

P4TG: 1 Tb/s Traffic Generation for Ethernet/IP Networks

STEFFEN LINDNER, MARCO HÄBERLE, (Student Member, IEEE), AND MICHAEL MENTH, (Senior Member, IEEE)
Chair of Communication Networks, University of Tuebingen, 72076 Tübingen, Germany
{steffen.lindner, marco.haeberle, menth}@uni-tuebingen.de

Corresponding author: Steffen Lindner (e-mail: steffen.lindner@uni-tuebingen.de).

This work was supported by the bwNET2020+ project which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg (MWK). The authors alone are responsible for the content of this paper

ABSTRACT In this work, we present P4TG, a P4-based traffic generator (TG) which runs on the programmable Intel Tofino™ ASIC. In generation mode, P4TG is capable of generating traffic up to 1 Tb/s split across 10x 100 Gb/s ports. Thereby it measures rates directly in the data plane. Generated traffic may be fed back from the output to the input ports, possibly through other equipment, to record packet loss, packet reordering, and sampled inter-arrival times (IATs) and round trip times (RTTs). In analysis mode, P4TG measures rates on the input ports, samples IATs, and forwards traffic through its output ports. We compare P4TG's performance with the one of the software TG TRex and the hardware TG EXFO. P4TG's code will be provided on GitHub.

INDEX TERMS P4, software-defined networks, traffic generation

LIST OF ABBREVIATIONS

ASIC	application-specific integrated circuit
CBR	constant bit-rate
CCDF	complementary cumulative distribution function
FPGA	field programmable gate array
IAT	inter-arrival time
ILP	integer linear program
L1	layer-1
L2	layer-2
L3	layer-3
MAT	match+action table
NIC	network interface card
P4	protocol-independent packet processors
RTT	round trip time
RX	reception
TG	traffic generator
TNA	Tofino Native Architecture
TX	transmission

I. INTRODUCTION

A traffic generator (TG) is a tool to generate or measure traffic in order to test network devices or applications. A TG may produce packets to approximate the behavior of real network traffic for specific use cases, e.g., some TGs mimic application layer conversations, or generate raw layer-2 (L2)/layer-3 (L3), i.e., Ethernet and/or IP, packets for stress-testing a network. TGs allow to configure the frame size and desired output rate and some can randomize certain packet headers, e.g., the IP destination field. Further, they support a wide range of network protocols, e.g., IP, MPLS, Ethernet, and some offer additional functionality such as means to debug and verify optical connectivity, e.g., transceiver power and bias. TGs also perform various measuring tasks. They typically measure L1 and L2 transmission (TX) and reception (RX) rates for generated traffic sent through the output port and for traffic received on the input port. For the latter, also round trip times (RTTs)[1], packet loss, out-of-order packets, as well as frame size and type are monitored.

TGs may be purely software-based or with hardware acceleration. Software-based TGs are flexible, provide many different features, and are customizable to individual needs. However, as they mostly run on general purpose CPUs, they only support low data rates and are prone to significant fluctuations in traffic generation [1]. In contrast, TGs with hardware acceleration support higher data rates and generate traffic more precisely. Unfortunately, they are not customizable and very expensive, up to tens of thousands of dollars

[1]In this context, the RTT is the time between the transmission of a packet on the out-port and its reception on the in-port.

for 100 Gb/s support. Therefore, they are not affordable for many research groups or smaller companies.

With the increasing prevalence of programmable switches, in particular, the Intel Tofino™ [2], it is possible to develop tools with extensive hardware support at an affordable price. In this paper, we present P4TG, a P4-based TG for Ethernet/IP networks using Intel Tofino™. P4TG measures L1 and L2 TX and RX rates, packet loss, out-of-order packets, round trip time (RTT), inter-arrival times (IATs), frame types, and frame sizes. Beside constant bit-rate (CBR) traffic generation, it also supports generation of random traffic. P4TG is capable of generating simultaneously up to 1 Tb/s split across 10x 100 Gb/s ports. This paper is structured as follows. Section II reviews related work and compares the presented TGs with P4TG. Then, we introduce P4 in Section III. Afterwards, we give an overview of P4TG in Section IV and explain its implementation in Section V. We evaluate P4TG in Section VI and discuss P4TG's limitations and possible future work in Section VII. Finally, we conclude the paper in Section VIII.

II. RELATED WORK

In this section, we first review related work on software-based traffic generation and hardware-based traffic generation. Then, we present work that compares software-based and hardware-based traffic generation and compare P4TG with the presented traffic generators.

A. SOFTWARE-BASED TRAFFIC GENERATION

MoonGen [3] is a highspeed packet generator based on DPDK. It is able to saturate a 10 Gb/s link with a single CPU core and can scale up to 120 Gb/s with multiple CPU cores. It further provides latency measurements including hardware timestamps with supported hardware NICs.

Pktgen-DPDK [4] and TRex [5] are software-based TGs that are also based on DPDK. Pktgen-DPDK is able to saturate a 10 Gb/s link with 64 byte frames per CPU core while TRex states to scale up to 200 Gb/s with 100 Mpps (mega-packets per second). In Section VI we compare P4TG with TRex.

Hock et al. [6] investigate how end-systems with general purpose CPUs need to be tuned to support TCP traffic generation at 100 Gb/s. They find that the placement of the generating application to a corresponding CPU socket in a multi-socket system heavily influences the achievable throughput. This may be the case when the network interface card (NIC) is connected to a different socket than the application. Their results may be applied to existing software-based traffic generators to improve performance.

B. HARDWARE-BASED TRAFFIC GENERATION

Yuan et al. [7] present an FGPA-based implementation for synthetic Ethernet traffic generation that can saturate a 10 Gb/s link. Their implementation is based on a COMBO-LXT FGPA board with two 10 Gb/s interfaces.

Plakalovic et al. [8] propose an affordable and extensible high-speed FGPA-based Ethernet TG that is able to fully utilize a 40 Gb/s link with 64 byte Ethernet frames. They developed and tested their solution on a DE10-Pro FGPA board which costs around 10k USD[2].

HyperTester [9] [10] combines software-based traffic generation and hardware-based traffic replication. Generated traffic from a CPU is replicated by the Intel Tofino™ to support higher data rates. Further, they propose a network testing API (NTAPI) that allows to express triggers for packet manipulation and statistic collection. Based on these expressions, template packets and a corresponding P4 program are generated. Although NTAPI may be used to express complex replication and measurement tasks, it is unclear whether complex descriptions can be supported on resource-constrained hardware targets. In contrast to HyperTester, P4TG does not rely on external traffic generation and further provides precise built-in measurement capabilities that can also be used to analyze external traffic. HyperTester's source code for the Intel Tofino™ is not publicly available.

Kundel et al. [11] [12] present P4STA, a framework for high performance packet timestamping and load generation for programmable network devices. They present an architecture where multiple load generation sources are aggregated and equipped with a precise hardware timestamp and sent to the device under test. When the device under test sends the packets back, they are equipped with a second timestamp and duplicated to an external host. There, the hardware timestamps are extracted and may be used to calculate RTTs and other metrics. However, it is unlikely that high data rates can be supported as the external host relies on software processing. P4STA supports limited integrated measurement capabilities, i.e., they only measure average delay and packet loss. Like HyperTester [9], P4STA relies on external traffic generation.

Commercial hardware-based traffic generators are sold by companies such as Spirent, Keysight Technologies, Exfo, Viavi, and others. While they offer high data rates with up to 400 Gb/s per port, they are very expensive and therefore not suited for academic research.

C. COMPARISON

Adeleke et al. [13] provide a survey of traffic generators used over the last 13 years. They examine the provided features and organize them into different categories. Their results show that the ten most used traffic generators in the literature are all software-based.

Botta et al. [14] and Emmerich et al. [1] evaluate a wide range of software-based TGs. While the investigated software TGs have many features, such as generation of arbitrary traffic patterns, they have precision problems for packet rates above 1 Mpps. Software-based TGs with hardware acceleration, e.g., MoonGen [3] or Pktgen-DPDK [4], provide higher

[2]As of 2022-08.

TABLE 1. Comparison of existing TGs with P4TG. ‡: With supported network cards.

	Software-based TGs			Hardware-based TGs			
Feature	MoonGen	Pktgen-DPDK	TRex	P4STA	HyperTester	EXFO FTB-1 Pro	P4TG
Internal traffic generation	X	X	X			X	X
Hardware timestamps	X[‡]	X[‡]	X[‡]	X	X	X	X
Nanosecond accuracy				X	X	X	X
100 Gb/s on several ports				X	X		X
Integrated measurement capabilities	X	X	X			X	X
Arbitrary traffic patterns	X	X	X				

data rates but still may significantly deviate from configured traffic patterns at high traffic loads.

In the following, we compare P4TG with several other TGs based on the following properties.

- **Internal traffic generation:** The capability to generate traffic without additional resources/servers.
- **Hardware timestamps:** Support for hardware timestamps with high accuracy.
- **Nanosecond accuracy:** Time-related events, e.g., timestamps, traffic generation, etc., are accurate up to a few nanoseconds. For instance, this ensures that CBR traffic has only little jitter.
- **100 Gb/s on several ports:** The capability to generate/replicate/aggregate 100 Gb/s traffic on several ports.
- **Integrated measurement capabilities:** The capability to perform measurement tasks, such as rate measurement, packet loss detection, out-of-order detection etc., without additional resources/servers.
- **Arbitrary traffic patterns:** The capability to generate arbitrary traffic patterns. It is assumed that not every traffic pattern can be supported by HyperTester's NTAPI.

Table 1 shows the comparison. P4STA and HyperTester rely on external traffic generation for testing purposes while P4TG and EXFO as well as all software-based TGs generate traffic without external help. All presented TGs support hardware timestamps although the software-based TGs require appropriate network interface cards (NICs). Only the hardware-based TGs, such as P4TG, P4STA, HyperTester, and EXFO FTB-1 Pro, provide nanosecond accuracy. Further, P4TG, P4STA, and HyperTester support generation of 100 Gb/s on several ports although P4STA and HyperTester rely on external traffic generation. EXFO FTB-1 Pro supports only traffic generation on a single 100 Gb/s port. Traffic generation capabilities of software-based TGs are limited by the available number of CPU cores, i.e., most are able to generate around 10 Gb/s per CPU core. Assuming a reasonable number of CPU cores in the order of 10 suffices to reach 100 Gb/s on a single port, but not on multiple ports. All considered TGs except P4STA and HyperTester provide integrated measurement capabilities. Finally, only software-based TGs are able to generate arbitrary traffic patterns.

III. INTRODUCTION TO P4

In this section we review the programming language P4. First, we give an overview of P4. Then, we explain P4 targets and the P4 pipeline. Finally, we describe match+action tables (MATs).

A. OVERVIEW

Protocol-independent packet processors (P4) [15] is a high-level programming language that describes the data plane of a P4-compatible device, so-called targets. A P4 program is mapped through a target-specific compiler to the pipeline of the device. Core components of P4 programs are programmable (de)parsers, match+action tables (MATs), and target-specific externs which can be used to program complex packet processing behavior. Available externs and P4 components are specified in a so-called P4 architecture that isolates the programmer from the low-level functionality of a device. Most P4 architectures include core P4 functionality such as multicast groups, packet cloning, and recirculation. Others may offer more specialized externs, e.g., complex math functions. Processing behavior can be changed during runtime by a control plane, e.g., by modifying rules of the MATs. P4 is applied in a wide range of use cases [16].

B. P4 TARGETS

A P4 target is a software or hardware platform that follows a certain P4 architecture, e.g., the software target BMv2 follows the v1model architecture whereas the Intel Tofino™ follows the Tofino Native Architecture (TNA) [17]. Software targets are implemented in high-level programming languages, e.g., C++. Therefore, their throughput is limited as they run on non-specialized hardware and packet processing is entirely done in software. Hardware targets perform packet processing in hardware and achieve high throughput. However, developing a P4 program for hardware targets is more challenging, as they typically have restrictions on the number of operations that can be applied per packet to ensure line rate processing. There are multiple types of P4 hardware targets, e.g., field programmable gate arrays (FPGAs), network interface cards (NICs), and application-specific integrated circuits (ASICs). Intel Tofino™ [2] is the only available[3] P4 programmable switching ASIC that can be programmed by end users.

[3]As of: 2022-11.

C. P4 PIPELINE

Each architecture defines its own so-called P4 pipeline. Figure 1 shows a simplified visualization of the P4 pipeline of the TNA [17] that is implemented by the Intel Tofino™. Each physical port can transmit and receive packets. Accordingly, Intel Tofino™'s P4 pipeline is divided into two different sections, *ingress* and *egress*. Incoming packets received by an input port are first processed by a programmable ingress parser. Thereby, packet headers are extracted and stored for later use within the ingress control, e.g., for matching in MATs. Afterwards, the packet is processed by the ingress control. The packet may be matched multiple times against user-defined MATs and header fields may be manipulated. During the ingress processing, the destination of the packet must be chosen, e.g., an egress port must be determined. At the end of the ingress section of the pipeline, the packet is serialized through the so-called ingress deparser that emits the headers according to the user defined ingress deparser. Afterwards, the packet is processed by the traffic manager. Thereby, the packet may be replicated depending on the specified destination, e.g., in the case of multicast. The packet is then enqueued for the selected egress port and later dequeued according to the underlying scheduling strategy of the Intel Tofino™ and sent to the egress parser. Egress parser, egress control and egress deparser perform similar operations as their ingress counterparts. Finally, the packet is transmitted through the specified output port or may be placed again in the ingress section of the P4 pipeline when the port is configured as recirculation port[4].

D. MATCH+ACTION TABLES (MATS)

A MAT performs packet-dependent actions by matching header fields and/or metadata against the entries of the MAT. Figure 2 illustrates the structure of a MAT.

When a packet is matched against a MAT, a so-called lookup key is formed. The lookup key consists of one or more header and/or metadata fields of the packet. Each component of the lookup key is compared to the stored key in the MAT according to a pre-defined *match type*. Three standard *match types* are defined in the P4 core library: exact, ternary, and longest prefix matching (lpm). Additional *match types* may be defined within specific P4 architectures. When a table entry matches the lookup key, the stored action is invoked. MATs are typically filled by a control plane.

IV. P4TG OVERVIEW

P4TG is a 1 Tb/s hardware-based TG for Ethernet/IP traffic[5] and is based on P4 and the Intel Tofino™ ASIC. P4TG supports two different modes, generation and analysis mode. In generation mode, P4TG generates either up to 7 different CBR streams or a single random traffic stream. Each stream can be configured with a traffic rate and frame size, and be assigned to up to 10 egress ports (output ports). Ethernet source and destination, IP source and destination, and IP ToS value can be set on a per port basis. Further, IP source and destination addresses can be randomized through a bitmask. P4TG supports 8 different L2 frame sizes by default, i.e., 64, 128, 256, 512, 1024, 1280 and 1518 bytes as recommended in RFC 2544 [18]. Other frame sizes can be added. Further, P4TG supports Jumbo frames with 9000 bytes.

In generation mode, P4TG measures for generated traffic per stream the TX and RX rates and the overall L1 and L2 TX and RX rate on a per port basis. Further, TX and RX packet rates are derived. The data are collected directly in the data plane with hardware timestamps for precise accuracy. Inter-arrival times (IATs) are sampled so that only a subset is evaluated by the control plane. For traffic sent on an output port and received on an input port, lost frames, out-of-order packets, frame types (unicast, multicast, broadcast) and frame sizes are also directly monitored on a per packet basis in the data plane leading to the highest possible precision. RTTs and IATs are sampled.

In analysis mode, P4TG acts as a transparent forwarder and similarly analyses external traffic received by an input port and forwards it to a specified output port. It measures L1 and L2 overall traffic rates, frame sizes, frame types and samples IATs. P4TG is configured with a web-based GUI that communicates through a REST-API with the control plane. Likewise statistics are inspected. Data plane, control plane, and the GUI are published at Github[6].

P4TG requires per output-input port pair two additional ports in loopback mode (see Section V for details). In our prototype, we utilize an Edgecore Wedge 100BF-32X [19] switch based on Intel Tofino™ 1 with 32 100 Gb/s ports. Therefore, our prototype is limited to 1 Tb/s traffic generation, i.e., 10 output-input port pairs[7] for traffic generation (10x 100 Gb/s = 1 Tb/s) and 20 additional ports in loopback mode. On larger Intel Tofino™ platforms, e.g., with 64 100 Gb/s ports, higher cumulated traffic rates may be achievable.

V. IMPLEMENTATION OF P4TG

We first introduce P4TG's monitoring concept to collect statistics and its traffic generation approach. We then explain the packet path along the ports in P4TG and its P4 processing pipeline. Finally, we summarize the statistics collection.

A. MONITORING CONCEPT

P4TG uses three different concepts to collect statistical data, DirectCounter-based, register-based, and sample-based monitoring. The stored statistics are retrieved through monitoring packets that are periodically generated (see Section V-B4).

1) DirectCounter-Based Monitoring

A MAT utilizes a DirectCounter extern to count how often a certain rule has matched. When matching on a certain frame

[4]A recirculation port is a port in loopback mode where packets transmitted through this port are immediately placed in the ingress section of the same port.

[5]Split across 10x 100 Gb/s ports.

[6]https://github.com/uni-tue-kn/P4TG

[7]Each of these 10 ports is used to transmit and receive traffic.

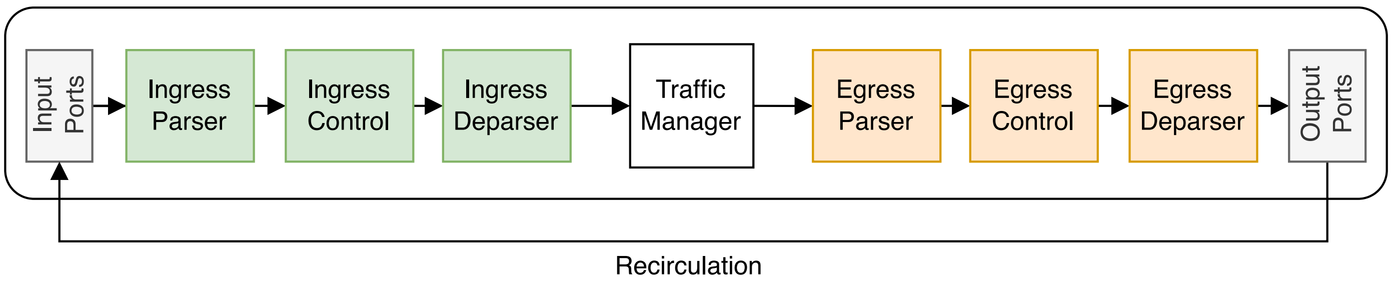


FIGURE 1. Simplified visualization of the P4 pipeline of the Tofino Native Architecture (TNA) [17] consisting of an ingress parser, ingress control, ingress deparser, traffic manager, egress parser, egress control, and egress deparser.

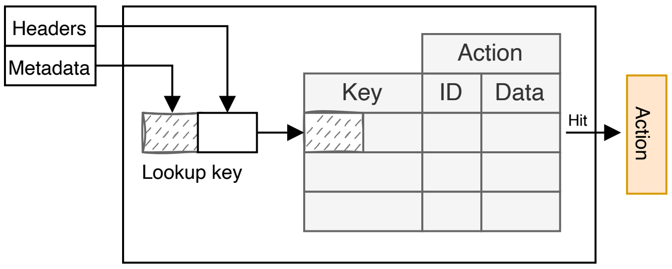


FIGURE 2. Structure of match+action tables (MATs) in P4. Illustration based on [16].

property, a DirectCounter records the number of packets with that property. P4TG utilizes this method to report frame types or frame sizes. The control plane polls these counters every 500 ms.

2) Register-Based Monitoring

Other metrics have aggregate semantics (e.g., overall sent and received bytes) or they are based on more context than the packet header (e.g., out-of-order count, lost count) so that DirectCounters are not applicable. P4TG collects such metrics by incrementing register fields. To report their content, P4TG generates a monitoring packet every 500 ms. When the data plane receives a monitoring packet, register fields and timestamp are copied into that packet. As a single packet cycle does not suffice to read all counters, monitoring packets are processed iteratively using recirculation before being sent to the control plane. Unlike DirectCounter-based monitoring, register-based monitoring maps measured results to exact intervals based on timestamps.

3) Sample-Based Monitoring

For some metrics like RTTs and IATs, average values do not suffice. Sample values are needed to compute standard deviation, minimum, maximum, and the distribution. However, reporting them per-packet to the control plane is too expensive. To reduce monitoring load, P4TG reports them to the control plane with a limited rate. This is done through a Meter extern on the Tofino which applies a three-color metering [20]. Thereby, the Meter extern ensures that samples are only sent to the control plane with a configured rate to prevent overloading.

B. TRAFFIC GENERATION

We first describe the built-in traffic generation capability of the Intel Tofino™ASIC [2]. Then, we explain various traffic generation methods of P4TG.

1) Tofino's Built-In Traffic Generation Capability

Intel Tofino™ comes with several pipes and each of them can be run with a separate P4 program. Every port is assigned to one of these pipes. Every pipe has an internal traffic generation port P_{TG}. Tofino allows to define up to 8 independent packet streams [17] for traffic generation and to explicitly activate them for for each pipe individually. Packet generation can be triggered through different events, i.e., a one-time timer, periodic timer, port-down event, or based on packet recirculation. P4TG leverages the periodic timer event to create traffic streams. Each stream is configured with the number of packets to be created in each period (burst), the packets' byte representation, a timeout in nanoseconds until the next packet is generated, and other means. P_{TG} generates packets with an additional 6 byte packet generation header that identifies the stream. Up to 100 Gb/s can be generated per pipe.

Although Tofino provides a powerful platform for data plane programming, care must be taken when implementing a program to not exceed available resources per packet cycle. Especially the use of registers and arithmetic operations are limited to facilitate line rate processing. The program code must fit into the available number of stages of the packet processing pipeline. This makes the implementation of P4TG challenging.

2) Traffic Generation with P4TG

The P4TG prototype is implemented on the Edgecore Wedge 100BF-32X switch [19] with 32 100 Gb/s ports based on Intel Tofino™ 1 which supports two pipes. We leverage both pipes for traffic generation. Packets are generated with an Ethernet, IPv4 and UDP header, as well as with a special 11 byte P4TG header as shown in Figure 3.

The P4TG header consists of a 32 bit sequence number for packet identification, a 48 bit transmission timestamp for RTT calculation, and an 8 bit ID for stream identification. All fields of the P4TG header are initialized within the P4 pipeline and their use is explained later. The overall header

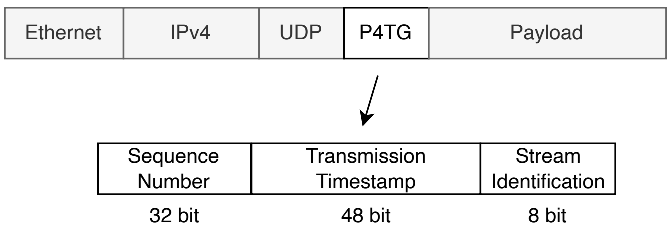


FIGURE 3. Packets are generated with an Ethernet, IPv4, UDP, and P4TG header.

size of a generated packet is 57 byte including 18 byte for Ethernet, 20 byte for IPv4, 8 byte for UDP, and 11 byte for P4TG. The remaining bytes (frame size - 57 byte) are padded with random bits. The minimum frame size is 64 byte. The L1 packet size includes a preamble and an inter-frame gap (additional 20 byte).

Generated packets are replicated to the configured egress ports using an internal multicast group. This allows the generation of up to 1 Tb/s, i.e., 100 Gb/s can be generated on up to 10 outgoing ports. Packet headers are rewritten on a per port basis so that different flows may be sent through each outgoing port.

In the following sections, we will describe P4TG's functionality for traffic generation on a single outgoing port. When generated packets are replicated to multiple outgoing ports, each outgoing port performs the described operations.

3) CBR Traffic Generation

P4TG provides two different modes for generation of CBR traffic: precision mode and rate mode. By default, P4TG is configured to use the rate mode.

a: Precision Mode

The target L1 traffic rate R^{L1}_{target} is given in Gb/s and the L2 frame size f^{L2}_{size} in byte. The periodic timer t is configured with nanoseconds according to Equation 1.

$$t = \left[\frac{(f^{L2}_{size} + 20) \cdot 8 \cdot 10^9}{R^{L1}_{target}} \right] \quad (1)$$

The resulting minimum IAT is rounded to entire nanoseconds as only integer values can be configured as timer values. After that time, a single L2 frame with f^{L2}_{size} byte is sent.

The advantage of the precision mode is smooth traffic at small time scales. Its disadvantage is that only certain traffic rates can be configured for given frame sizes due to the integer-based timer. For instance, 50 Gb/s L2 Ethernet traffic with 64 byte frames requires packet generation every 10.24 ns. As the periodic timer is configured as integer, packets are generated every 10 ns, which results in 51.2 Gb/s. Moreover, when very large data rates and very small frame sizes are configured, the expected data rate is not fully achieved (see Section VI-A).

b: Rate Mode

In rate mode, n packets are sent every t nanoseconds, e.g., $n = 25$ packets are sent every $t = 256$ nanoseconds to better achieve 50 Gb/s on L1 with 64 byte L2 frames. The minimum integer timeout t in nanoseconds and the burst size n for this mode can be derived from the following integer linear program (ILP) shown in Equation 2.

$$\begin{array}{ll} \text{minimize} & \left| R^{L1}_{target} \cdot t - (n \cdot (f^{L2}_{size} + 20) \cdot 8) \right| \\ \text{subject to} & t \in \mathbb{N}^+ \text{ and } n \in \mathbb{N}^+ \end{array} \quad (2)$$

The solution of this ILP yields the desired IAT t and burst size n. An additional upper bound for n can limit the burst size of the generated traffic but possibly reduces the rate accuracy [8]. As the traffic generation port may not be able to fully achieve the configured rate, P4TG generates $R^{L1}_{target}/2$ on both pipes when data rates larger than 75 Gb/s are desired.

4) Generation of Monitoring Packets

Monitoring packets are generated by a CBR stream with a monitoring header as shown in Figure 4.

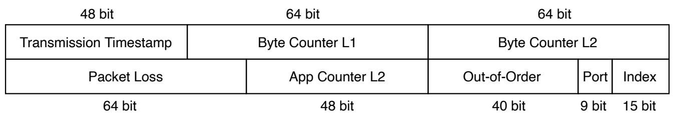


FIGURE 4. Monitoring packets are generated with a monitoring header that is used for statistic collection.

The monitoring header consist of a 48 bit timestamp, a 64 bit byte counter for L1 traffic, a 64 bit byte counter for L2 traffic, a 64 bit counter for lost packets, a 48 bit byte counter for L2 traffic of a specific stream, a 40 bit counter for out-of-order packets, a 9 bit port identification number, and a 15 bit index field that is used to access the stored L2 counter for the requested stream. During statistic collection (see Section V-E) the fields of the monitoring packet are filled with the respective stored statistics for the given port. The generation of monitoring packets requires one out of 8 configurable streams. Therefore, only 7 other streams can be simultaneously configured for traffic generation.

5) Random Traffic Generation

P4TG also supports generation of random traffic. First, traffic is generated at maximum rate, which results in a minimum IAT t_{min}. t_{min} depends on the desired frame size. Then, frames are forwarded only with probability p. It is computed as shown in Equation 3.

$$p = R^{L1}_{target} / \frac{(f^{L2}_{size} + 20)}{t_{min}} \quad (3)$$

[8]Let a be the desired relative rate accuracy and let $i = \frac{f^{L2}_{size} + 20 \text{ byte}}{R^{L1}_{target}}$ be the desired IAT in ns (may be a fractional number). Then, $d = \frac{i - \lfloor i \rfloor}{i}$ is the relative deviation from the target rate. Using $\frac{d}{a}$ as upper bound on n yields the desired accuracy a. In our study we utilize $a = 0.001$.

This leads to a geometric distribution of the IATs which is the discrete approximation of the exponential distribution. Thus, random traffic approximates Poisson arrivals. As this generation method requires 100 Gb/s, no additional CBR stream can be produced except for monitoring packets.

C. PACKET PATH

When a packet is received, it is placed in the *ingress* section of the corresponding port. The packet's egress port is determined in this ingress section. Based on this port, the packet is internally forwarded to the *egress* section of the sending port.

P4TG has up to 10 output ports P^i_{out} to send generated traffic and up to 10 input ports P^j_{in} to receive traffic to be measured. In the following we will refer to them as P_{out} and P_{in} without superscript for the sake of readability. In addition, per output-input port combination it utilizes two additional ports in loopback mode for traffic recirculation: P^R_{out} and P^R_{in}. They are needed for iterative processing of monitoring packets and for statistic collection of generated or measured traffic[9]. P_{TG} is the internal traffic generation port.

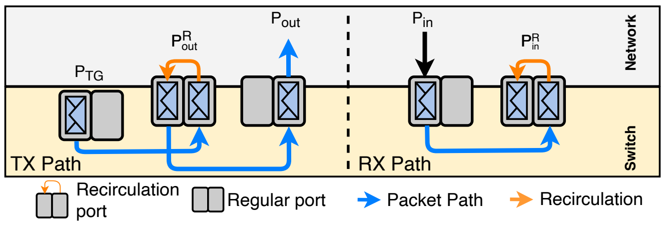


FIGURE 5. Packet path with P4TG.

Figure 5 illustrates the packet path for a single output-input port pair through the mentioned ports. Generated packets follow the TX path. A packet is generated by P_{TG} and internally forwarded from P_{TG}'s ingress section to (possibly multiple) P^R_{out}'s egress section. The loopback forwards the packet to the ingress section of the same port. From there, it is internally forwarded to the egress section of P4TG's corresponding output port P_{out}. Generated packets with a P4TG header may be forwarded via external equipment to P_{in} to check RTTs, IATs, out-of-order packets and packet loss. After reception by P_{in} they follow the RX path. When a packet is received by the ingress of P_{in}, it is internally forwarded to the egress section of P^R_{in} and afterwards via loopback to its ingress section.

In analysis mode, packets are received via P^R_{in} and forwarded to P^R_{out} to meter rates and sample IATs. This may be done on up to 10 different P^R_{in}-P^R_{out} pairs.

Monitoring packets are generated by P_{TG}, forwarded to (possibly multiple) P^R_{out}, repeatedly filled with counter information and recirculated to P^R_{out} [10], then forwarded to P^R_{in}, repeatedly filled with counter information and recirculated. After each recirculation, collected statistics of the monitoring packets are sent to the control plane[11].

[9]The public TNA [17] states that some packet characteristics, such as the packet length, can only be accessed in the *egress* section. However, for statistic collection purposes, P4TG needs to access the packet length without actually sending the packet through an egress port.

[10]Recirculation is needed as a single pipepline iteration does not suffice to collect all register values.

D. P4TG PIPELINE

P4TG is implemented in both the *ingress* and *egress* section of Tofino's pipeline. For ease of explanation, we first consider the egress section and then the ingress section.

1) Egress Section

Figure 6 illustrates the *egress* section of P4TG.

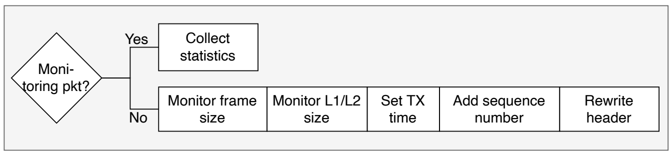


FIGURE 6. Behavior of P4TG's *egress* section.

P^R_{out} and P^R_{in} process monitoring packets and take care of statistics collection. If the packet is a monitoring packet, counter information is collected and control information is set in the metadata. After recirculation, a digest is created to report the collected information to the control plane. For other packets, the L2 frame sizes are monitored through a DirectCounter, and the L1 and L2 frame sizes are accumulated in registers. Finally, the TX timestamp and sequence number are set in the P4TG header before the packet is transmitted. Ethernet and IPv4 header are also rewritten according to the applied configuration which may include randomization.

The egress section is the same for P_{TG}, P_{in}, and P_{out} although the functionality is not needed. However, it does not interfere and leads to a simpler program. Moreover, resetting the TX timestamp in P_{out} avoids recirculation delay for RTT measurement.

2) Ingress Section

Figure 7 illustrates the *ingress* section of P4TG. It implements the behavior of P_{TG}, P_{in}, P^R_{out} and P^R_{in}.

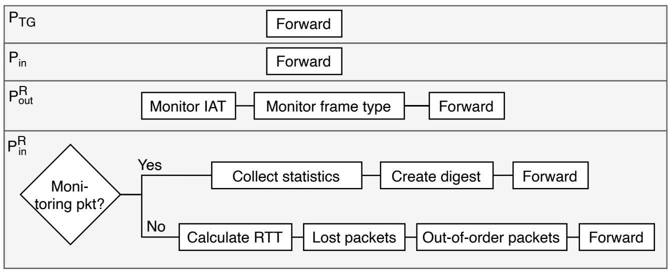


FIGURE 7. Port-specific behavior of P4TG's *ingress* section.

[11]Monitoring packets require at most 25 Kb/s on L1.

The ingress sections of P_{TG} just sets the P_{out}^{R} as outgoing port[12], and the ingress section of P_{in} just sets the P_{in}^{R} as outgoing port. The ingress section of P_{out}^{R} calculates its IAT and possibly reports it to the control plane, and it monitors the frame type using a DirectCounter. Then the packet is forwarded to the *egress* section using a table which is set by the control plane depending on generation or analysis mode. The ingress section of P_{in}^{R} processes monitoring packets, calculates statistics for received packets, and forwards these packets accordingly. For other packets, the RTT is calculated, out-of-order is checked, and the number of lost packets are computed. Those metrics are accumulated in registers. Finally, the packet is forwarded in analysis mode to P_{out}^{R}.

E. STATISTICS COLLECTION

We explain how individual statistics are collected.

1) Frame Types and Frame Sizes

P4TG monitors frame types, i.e., unicast, multicast, broadcast, and frame sizes directly in the data plane using DirectCounter-based monitoring. This is done for generated and received packets by P_{out}^{R} and P_{in}^{R}, respectively. P4TG maintans a match-action table *frame_type* in the *ingress* section that matches on the IPv4 destination address using an lpm match type and the ingress port using an exact match. The *frame_type* table has entries that match on the multicast address space (224/4), the broadcast address space (depending on the configuration), and a default match entry (for general unicast packets). Additionally, P4TG maintains a P4 table *frame_size* in the *egress* section that matches on the packet size using a range match and on the egress port using an exact match. The *frame_size* table has entries for ranges (0, 63), (64, 64), (65, 127), (128, 255), (256, 511), (512, 1023), (1024, 1518), (1518, 10000).

2) TX and RX Packet and Data Rates

Sent and received bytes on L1 and 2 are accumulated in registers in the egress section of P_{out}^{R} and P_{in}^{R}. Two 64 bit registers are leveraged to store a running sum of L1 and L2 packet sizes. As the TNA [17] can only store 32 bit per register entry, we build a 64 bit register by using three 32 bit registers, one register to store the lower 32 bit, one register to store the higher 32 bit, and a 32 bit register to calculate and store the carry bit when an overflow occurs in the lower 32 bit. For a packet on egress port P with Ethernet frame size f_{size}^{L2}, the running sum is calculated as shown in Equation 4 and Equation 5.

$$R_{meter}^{L1}[P] = R_{meter}^{L1}[P] + f_{size}^{L2} + 20 \tag{4}$$

$$R_{meter}^{L2}[P] = R_{meter}^{L2}[P] + f_{size}^{L2} \tag{5}$$

Thus, $R_{meter}^{L*}[P_{out}^{R}]$ accumulates transmitted data and $R_{meter}^{L*}[P_{in}^{R}]$ accumulates received data[13].

[12]The packet may be replicated to multiple P_{out}^{R} using an appropriate multicast group.

[13]The port number P_{out}^{R} and P_{in}^{R} are used as index to the register.

The registers are regularly read by monitoring packets together with a timestamp and delivered to the control plane. The difference in bytes of consecutive monitoring packets divided by the difference of their timestamps yields the data transmission/reception rate in Gb/s. The number of packets within an interval is the difference of L1 and L2 data volume for that interval divided by the L1 header size (20 byte). This allows the computation of the packet rate in Mpps for that interval. To cope with unstable rates, the interval-based rates are smoothed with a configurable memory (see TDRM-DTWMA-UEMA [21]).

3) Lost and Out-of-Order Packets

When a packet enters the egress section of P_{out}^{R}, a sequence number is set in the P4TG header. The sequence number is incremented by 1 and stored in a 32 bit register. Lost packets and out-of-order packets are computed and stored by P_{in}^{R}. To that end, the *ingress* section of P_{in}^{R} stores the next expected sequence number in a 32 bit register[14]. When the ingress section receives a packet with a higher sequence number than expected, the difference in sequence number is taken as number of lost packets. This number is accumulated in a 64 bit register[15]. When the ingress section receives a packet with a lower sequence number than expected, it is assumed that this packet is out-of-order and a corresponding 64 bit register is increased by one.

With this approach, each out-of-order packet is also counted as lost packet (however, not every lost packet is counted as out-of-order). Therefore, the control plane reports the difference[16] between lost and out-of-order packets as lost packets.

4) Round Trip Time

When a packet is transmitted through P_{out}, a 48 bit timestamp t_{TX} is set in the P4TG header. When the packet is received, its RTT is computed in the ingress section of P_{in}^{R} by the elapsed time since t_{TX}[17].

That value is reported to the control plane according to its configured meter instance which provides a sampled statistic (see Section V-A3).

5) Inter-Arrival Times (IATs)

IATs are measured in the ingress section of P_{out}^{R} for generated traffic and in the ingress section of P_{in}^{R} for measured traffic. A register stores the last packet arrival instant[18]. When a packet is received, the time since the last packet arrival is taken as IAT and the current time is stored as last packet

[14]Wrap-around of sequence numbers is appropriately handled.

[15]This 64 bit register is again composed of three 32 bit registers.

[16]The control plane reports $max(0, \text{lost} - \text{out-of-order})$ as lost packets.

[17]Wrap-around problems with timestamps are avoided by ignoring RTTs larger than a day.

[18]The last packet arrival is stored in a 48 bit register. It is composed of a 32 bit (lower bits) and 16 bit (higher bits) register. 48 bit are sufficient to avoid wrap-around for 4 days. In case of wrap-around, IATs are larger than one day. The control plane ignores such IATs.

arrival. This IAT is sent according to its meter instance to the control plane (see Section V-A3).

VI. EVALUATION OF P4TG

In this section, we evaluate the accuracy of P4TG and compare it to the one of other TGs. We utilize P4TG on the Intel Tofino™ 1 platform. We run the software TG TRex [5] on a server with 16 GB RAM and 10 Intel(R) Xeon(R) Gold 6134 CPU cores with CPU pinning and a 100 Gb/s Mellanox Connectx-5 NIC. We further utilize the hardware TG EXFO FTB-1 Pro [22]. In the following, we evaluate traffic generation on a single port of P4TG. However, due to the multicast replication, P4TG is capable of generating up to 1 Tb/s, i.e., 100 Gb/s on up to 10 ports.

A. RATE ACCURACY OF GENERATED CBR TRAFFIC

We configure the TGs to generate CBR traffic at different rates R^{L1}_{target} and with different frame sizes f^{L2}_{size}. We measure the generated rates with P4TG. Table 2 indicates for different parameter combinations to what percentage the measured rate achieves the target rate.

TABLE 2. Measured L1 rate relative to target rate R^{L1}_{target}.

Gen. method	f^{L2}_{size} (byte)	R^{L1}_{target} (Gb/s)				
		0.001	10	50	75	100
P4TG PM (1 Pipe)	64	100.00	98.80	91.10	99.37	81.98
	256	100.00	99.80	96.18	97.06	92.89
P4TG PM (2 Pipe)	64	100.00	98.8	93.72	99.37	91.09
	256	100.00	99.80	97.96	97.06	94.52
P4TG RM (1 Pipe)	64	100.00	100.00	99.38	99.37	81.98
	256	100.00	99.80	99.76	99.77	99.77
P4TG RM (2 Pipe)	64	100.00	100.00	99.38	99.37	99.98
	256	100.00	99.80	99.92	99.77	99.92
TRex	64	99.99	100.00	100.00	91.30	68.48
	256	100.00	99.96	99.94	83.66	61.92
EXFO	64	100.00	100.00	100.00	100.00	100.00
	256	100.00	100.00	100.00	100.00	100.00

When P4TG is configured with precision mode (PM) and a single pipe for traffic generation, the achieved relative rate decreases with increasing target rate. Only 81.98 Gb/s are achieved instead of 100 Gb/s. There are 3 reasons. First, frames are generated with a 6 byte internal header which is removed before sending the packet. This limits the traffic rate from a single pipe to 93.3% when a rate close to 100 Gb/s is desired. Second, as only an integral IAT can be configured, PM utilizes an IAT of 7 ns instead of 6.72 ns. This reduces the rate to 96% and effects that relative rates are not strictly decreasing. Third, when P4TG is under high load, some packets may be generated late. Due to constant IATs P4TG cannot compensate for that delay, which leads to a reduced traffic rate. Larger packets (256 byte) reduce the first and third problem (92.89 Gb/s), two pipes instead of one reduce the third problem (91.09 Gb/s).

P4TG with rate mode (RM) solves the second problem if packets are at least 256 bytes large. P4TG with RM and two pipes meets the target rate even more precisely, in particular 100 Gb/s can be generated with 64 byte frames. Thus, we recommend to use P4TG in RM with 2 pipes unless very constant IATs are needed.

TRex has major problems to achieve the target rates at lower speeds and its output seems to be limited below 70 Gb/s. This problem gets worse with larger frames (256 byte).

The hardware TG EXFO generates the target rate very accurately for all considered frame sizes and traffic rates.

B. ACCURACY OF MEASURED IATs

P4TG analyzes statistical properties of IATs based on a sampled subset. We have no other tool to measure the distribution of IATs. However, we have mean rates measured with EXFO and P4TG from which we derive the exact mean of IATs. We observe that the mean of sampled IATs is larger than the exact one (6.89 ns vs. 6.76 ns) when the traffic load in terms of packets is very high (64 byte frames and 100 Gb/s). Under all other conditions (frames sizes 256 byte or rates up to 75 Gb/s) the means are identical. Therefore, we trust in the sampled set of IATs so that we use them to derive other statistical measures like standard deviation and distribution.

C. IAT ANALYSIS OF GENERATED TRAFFIC

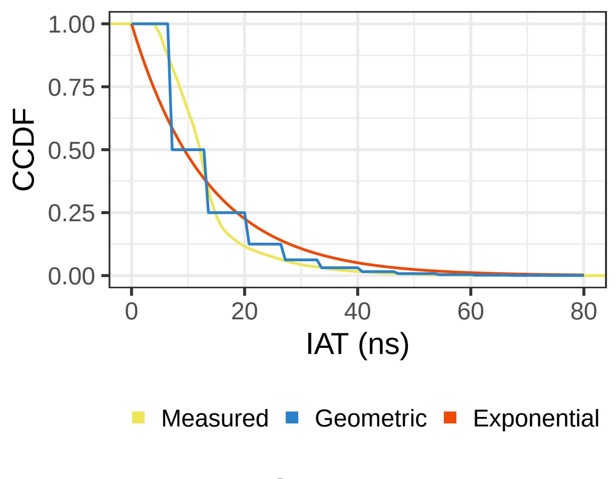
We first consider CBR traffic and then random traffic.

1) CBR Traffic

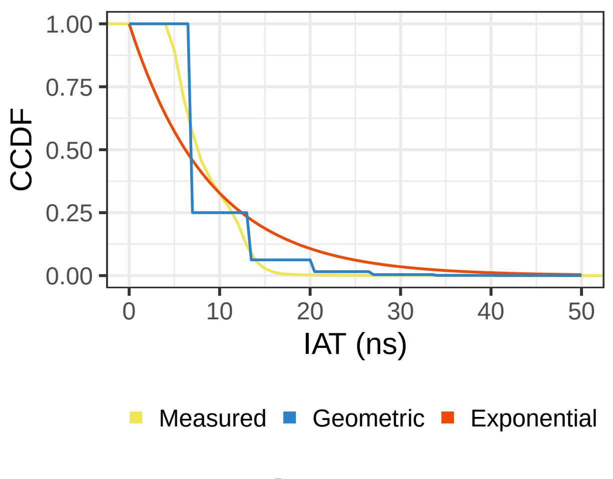
We compute the standard deviation of IATs sampled by P4TG. They are compiled in Table 3 for various generation methods and traffic rates. P4TG in PM with 1 pipe produces the least standard deviations. However, this method has problems to achieve high rates with frame sizes. P4TG in RM with 2 pipes causes larger standard deviations as $n-1$ small IATs are followed by one large IAT. The standard deviations of TRex traffic are several orders of magnitude larger than those of P4TG. Also the standard deviations of EXFO's traffic are an order of magnitude larger than those of P4TG if the target rate is small, i.e., the IATs are long. Thus, neither TRex nor EXFO generate constant IATs, they apparently use large and short IATs to adaptively meet the target rate.

2) Random Traffic

Figures 8(a) and 8(b) show the complementary cumulative distribution function (CCDF) of IATs from P4TG's random traffic derived from 500.000 samples. They are compared to geometric CCDFs which are step-functions and the underlying model of the generation method for random traffic. The two CCDFs are in good accordance and the match becomes better for smaller traffic rates. Deviations are due to jitter in the generation method at a time scale of a very few ns. The IATs of both CCDFs are bounded by some minimum. This is in contrast to the CCDF of exponentially distributed



(a) R^{L1}_{target} = 50 Gb/s



(b) R^{L1}_{target} = 75 Gb/s

FIGURE 8. CCDF of IATs in random traffic with 64 byte frames: measured, theoretic, and approximated by exponential IATs.

TABLE 3. Standard deviation based on sampled IATs for 64 byte frames.

Gen. method	R^{L1}_{target} (Gb/s)	$\sigma(IAT)$	R^{L1}_{target} (Gb/s)	$\sigma(IAT)$
P4TG PM (1 Pipe)	0.1	2.47 ns	50	2.51 ns
	1	2.66 ns	75	3.21 ns
	10	2.21 ns	(100	2.90 ns)
P4TG RM (2 Pipe)	0.1	4.81 μs	50	4.23 ns
	1	51.72 ns	75	3.24 ns
	10	84.38 ns	100	1.36 ns
TRex	0.1	36.8 μs	50	31.62 ns
	1	4.58 μs	(75	13.89 ns)
	10	463.62 ns	(100	13.89 ns)
EXFO	0.1	21.65 ns	50	5.65 ns
	1	24.85 ns	75	3.23 ns
	10	31.47 ns	100	1.25 ns

IATs which make up Poisson traffic. However, due to minimum frame sizes smaller IATs are technically not feasible. Therefore, P4TG's random traffic is a good approximation of Poisson traffic for experimental purposes.

VII. LIMITATIONS & FUTURE WORK

We first summarize P4TG's limitations. Then we discuss future work.

A. LIMITATIONS

P4TG can be configured with up to 7 different traffic stream definitions (frame size and target rate) as the TNA [17] allows the definition of up to 8 independent packet streams. Consequently, P4TG cannot be used to generate arbitrary traffic patterns. However, more than 7 different packet streams can be generated by randomizing or rewriting packet headers on a per-port basis. Further, P4TG is limited to 1 Tb/s traffic generation on our prototype although the underlying hardware is capable of processing 3.2 Tb/s. This is due to the fact that additional recirculation ports are needed for measurement purposes. However, other hardware-based TGs, such as the EXFO FTB-1 Pro [22], have significantly higher acquisition costs but generate only 100 Gb/s.

B. FUTURE WORK

Future work could extend P4TG to perform automated network testing as proposed in RFC 2544 [18] or ITU-T Y.1564 [23]. These tests are typically applied by network operators to verify service level agreements. In addition, P4TG can be evaluated on Tofino™ 2 and 3 platforms to test traffic generation at up to 400 Gb/s per port.

VIII. CONCLUSION

In this paper, we presented P4TG, a TG based on the P4-programmable Intel Tofino™ ASIC. In generation mode, P4TG generates up to 1 Tb/s, i.e., 100 Gb/s on 10 ports, Ethernet traffic and supports packet customization. It measures rates, frame types and sizes, packet loss, and out-of-order packets of generated and received traffic in the data plane and samples RTTs and IATs in the control plane. We described P4TG's implementation in detail and compared its performance with other TGs. P4TG is able to produce 64 byte frames CBR traffic at 100 Gb/s with stabler IATs than other TGs. Moreover, P4TG is able to produce random traffic. In analysis mode, P4TG analyses external traffic, measures rates, frame types and sizes, and samples IATs. It further acts as transparent forwarder such that it can analyze traffic without disrupting connectivity.

P4TG comes with clearly lower costs than other hardware-based commercial hardware solutions. Further, with the Intel Tofino 2 and Intel Tofino 3, P4TG may be able to generate traffic at even higher rates (up to 400 Gb/s per port).

REFERENCES

[1] Paul Emmerich, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle. Mind the Gap - A Comparison of Software Packet Generators. In ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pages 191–203, 2017.

[2] Intel. Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, 2021. Accessed on 21.01.2023.
[3] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In ACM Internet Measurements Conference (IMC), pages 275–287, 2015.
[4] Keith Wiles. The pktgen application. https://pktgen-dpdk.readthedocs.io/en/latest/, 2010. Accessed on 11.08.2022.
[5] TRex Team. TRex - Realistic Traffic Generator, 2022. Accessed on 17.08.2022.
[6] Mario Hock, Maxime Veit, Felix Neumeister, Roland Bless, and Martina Zitterbart. TCP at 100 Gbit/s – Tuning, Limitations, Congestion Control. In IEEE Conference on Local Computer Networks (LCN), pages 1–9, 2019.
[7] DongMing Yuan, Wei Yi, HeFei Hu, and XiaoDong Shi. A fast, affordable and extensible FPGA-based synthetic Ethernet traffic generator for network evaluation. In IEEE International Conference on Computer and Communications (ICCC), pages 1036–1040, 2017.
[8] Matej Plakalovic, Enio Kaljic, and Miralem Mehic. High-Speed FPGA-Based Ethernet Traffic Generator. In International Conference on Information, Communication and Automation Technologies (ICAT), 2022.
[9] Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, and Jianping Wu. HyperTester: High-Performance Network Testing Driven by Programmable Switches. In ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), page 30–43, 2019.
[10] Dai Zhang, Yu Zhou, Zhaowei Xi, Yangyang Wang, Mingwei Xu, and Jianping Wu. HyperTester: High-Performance Network Testing Driven by Programmable Switches. IEEE/ACM Transactions on Networking, 29(5):2005–2018, May 2021.
[11] Ralf Kundel, Fridolin Siegmund, Jeremias Blendin, Amr Rizk, and Boris Koldehofe. P4STA: High Performance Packet Timestamping with Programmable Packet Processors. In IEEE/IFIP Network Operations and Management Symposium (NOMS), pages 1–9, 2020.
[12] Ralf Kundel, Fridolin Siegmund, Rhaban Hark, Amr Rizk, and Boris Koldehofe. Network Testing Utilizing Programmable Network Hardware. IEEE Communications Magazine, 60(2):12–17, 2022.
[13] Oluwamayowa Ade Adeleke, Nicholas Bastin, and Deniz Gurkan. Network Traffic Generation: A Survey and Methodology. ACM Computing Surveys, 55(2), January 2022.
[14] Alessio Botta, Alberto Dainotti, and Antonio Pescape. Do You Trust Your Software-Based Traffic Generator? IEEE Communications Magazine, 48(9):158–165, 2010.
[15] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. ACM SIGCOMM Computer Communication Review, 44(3), 2014.
[16] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with P4: Fundamentals, advances, and applied research. Journal of Network and Computer Applications, 212:103561, 2023.
[17] Intel. P416 Intel Tofino Native Architecture - Public Version. https://github.com/barefootnetworks/Open-Tofino, 2021. Accessed on 17.08.2022.
[18] S. Bradner and J. McQuaid. RFC2544: Benchmarking Methodology for Network Interconnect Devices, March 1999.
[19] Edge-Core Networks. Wedge100BF-32X/65X Switch. https://www.edge-core.com/_upload/images/Wedge_100-32X_DS_R04_20170615.pdf, 2021. Accessed on 21.01.2023.
[20] J. Heinanen and R. Guerin. RFC2697: A Two Rate Three Color Marker, September 1999.
[21] Michael Menth and Frederik Hauser. On Moving Averages, Histograms and Time-Dependent Rates for Online Measurement. In ACM/SPEC International Conference on Performance Engineering (ICPE), April 2017.
[22] EXFO. FTB-1v2/FTB-1 Pro Platform. https://www.exfo.com/umbraco/surface/file/download/?ni=10900, 2019. Accessed on 17.08.2022.
[23] ITU-T Recommendation Y.1564 : Ethernet service activation test methodology, 2016.

STEFFEN LINDNER is a Ph.D. student at the chair of communication networks of Prof. Dr. habil. Michael Menth at the Eberhard Karls University Tuebingen, Germany. He obtained his master's degree in 2019 and afterwards, became part of the communication networks research group. His research interests include software-defined networking, P4 and congestion management.

MARCO HAEBERLE, (Student Member, IEEE) studied computer science at the University of Tuebingen, Germany, and received his Master's degree. Since then, he has been a researcher at the Chair of Communication Networks at the University of Tuebingen, pursuing his PhD. His main research interests include software defined networking, P4, network security, and service function chaining.

MICHAEL MENTH, (Senior Member, IEEE) is professor at the Department of Computer Science at the University of Tuebingen/Germany and chairholder of Communication Networks since 2010. He studied, worked, and obtained diploma (1998), PhD (2004), and habilitation (2010) degrees at the universities of Austin/Texas, Ulm/Germany, and Wuerzburg/Germany. His special interests are performance analysis and optimization of communication networks, resilience and routing issues, as well as resource and congestion management. His recent research focus is on network softwarization, in particular P4-based data plane programming, Time-Sensitive Networking (TSN), Internet of Things, and Internet protocols. Dr. Menth contributes to standardization bodies, notably to the IETF.

•••

2 Accepted Manuscripts (Additional Content)

2.1 P4 In-Network Source Protection for Sensor Failover

P4 In-Network Source Protection for Sensor Failover

Steffen Lindner*, Marco Häberle*, Florian Heimgaertner*, Naresh Nayak†, Sebastian Schildt†, Dennis Grewe†, Hans Loehr†, and Michael Menth*
* University of Tuebingen, Chair of Communication Networks, Tuebingen, Germany
Email: {steffen.lindner,marco.haeberle,florian.heimgaertner,menth}@uni-tuebingen.de,
† Corporate Sector Research and Advance Engineering, Robert Bosch GmbH, Renningen, Germany
Email: {naresh.nayak,sebastian.schildt,dennis.grewe,hans.loehr}@de.bosch.com

***Abstract*—Automated systems like industrial applications or autonomous cars heavily rely on sensor information. To increase reliability, several sensors may be used to provide identical data, e.g., temperatures or velocity. Applications exploiting this data may either use both data streams or rely on a single primary data stream until the primary stream fails. This increases the complexity of the application and is prone to errors. In this paper we present a prototype and mechanisms for in-network sensor failover. Our novel prototype detects the failure of a primary sensor and delivers in turn the data of a redundant sensor to the application.**

I. Introduction

Reliability is an important property of system critical infrastructure. Traditional resilience mechanisms protect against single link and single node failure, i.e. the connectivity can be restored if a single link or a single node fails. The detection of a link or node failure may require a few 10s of milliseconds so that traffic loss cannot be avoided. Loop-Free Alternates (LFAs) [1] are an example for such a mechanism. To respond faster to an incident, traffic can be transmitted redundantly on multiple paths. If there is an error on one path, the traffic on another path is still transmitted correctly. As traffic is transmitted redundantly, a substantially higher bandwidth is required and the resources of the network are not used in an optimal manner. 1+1 protection [2] is an example for a redundant protection mechanism. These network protection mechanisms protect only against network failures like link or node failures. However, they cannot help when the source node fails. The source can be protected by having several sources providing the same information. Possible use cases include sensors in industrial facilities, (autonomous) cars, or other publish/subscribe scenarios. In a publish subscribe environment, publishers offer data that can be accessed by subscribers. To compensate for the failure of a publisher, several publishers provide the same information. In case of an error, the information is provided by another publisher. To that end, the failure of a publisher must be detected and the network re-configured. Alternatively, an application can subscribe to several streams and independently perform error handling. These methods either require additional signaling effort, and thus time, or involve redundant implementations in different applications. In industrial and time-critical networks in general, the demands on data streams are usually stringent, not only in terms of bandwidth, latency and jitter, but also in terms of response time in the event of errors. While new paradigms and standardization efforts like time-sensitive networking (TSN) and deterministic networking (DetNet) provide a broad feature set to guarantee these requirements, they come with the need of specialized and costly hardware. With software defined networking, new mechanisms and protocols can be developed without requiring specialized hardware. Technologies such as P4 enable the data plane of a compatible switch to be programmed, paving the way for new prototypes and mechanisms. In this paper, we present two mechanisms that can be used to implement a fast failover for redundant sensor pairs in a network. In the error-free case, only data from the primary sensor is forwarded. If the primary sensor fails, the switch detects the missing data of the primary sensor and forwards the data of the redundant sensor. The presented mechanisms neither require additional signaling nor the reconfiguration of the network. Furthermore, the failover is transparent for the receiver. We present a P4-based implementation of the proposed mechanisms and evaluate them on the high-performance P4 switching ASIC Tofino.

The remainder of the paper is structured as follows. We discuss some related resilience mechanisms in Section II. Section III gives an overview of the data plane programming language P4. Afterwards, we introduce two mechanisms for in-network sensor failover in Section IV. Section V gives some insights regarding the P4-based implementation of the proposed mechanisms. We evaluate and discuss the mechanisms in Section VI. Finally, Section VII concludes the paper.

The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/1-2. The authors alone are responsible for the content of the paper.

II. Related Work

In this section we present some related resilience mechanisms. Closely related to our case study is the area of publish/subscribe networks. In a publish/subscribe system, users can subscribe to publishers to receive information. To ensure reliability, several publishers can provide the same information. In our deployment scenario the user corresponds to the application and the sensors correspond to publishers.

In [3], the authors propose an extension to the PURSUIT framework to introduce source recovery in information-centric networks (ICN). They introduce a new component, called resilience management (RM). To detect a component failure, each node exchanges link state updates (LSU) with its neighbors. If a node does not receive a LSU within a time limit, a link failures is assumed. The topology management (TM), another component in the PURSUIT framework, receives this information and updates the topology accordingly. Afterwards, affected distribution trees can be altered and the connectivity can be restored. If the failure affects a publisher, the distribution trees can be adjusted to use an alternative publisher. Hoefling et. al [4] propose a distributed load balancing mechanism for SeDAX, a publish/subscribe information-centric networking architecture. They ensure robustness by replicating content on multiple nodes. If the primary node fails, it is possible to switch directly to the second node in order to continue offering the content.

The data distribution service (DDS) [5] is a platform-independent standard for data-centric publish subscribe systems. It is designed for real-time systems with low latency and high robustness requirements. DDS facilitates that several publishers supply the same data while a subscriber always receives data from the so-called ”most-trusted” publisher. If the ”most-trusted” publisher fails, the application automatically uses the data of the next publisher. This failure mechanism requires an action from the application, provided by the DDS framework. Campelo et. al [6] propose an architecture for a fault-tolerant distributed industrial control system composed of several micro-controllers. The system can switch to an alternative micro-controller in case of a failure. Another architecture for safety-critical applications is described in [7].

III. P4 Foundations

We first give an overview of P4. Then we summarize basics of the P4 pipeline that are needed to understand the implementation of the P4 implementation.

A. P4 Overview

P4 is a programming language for protocol-independent packet processors [8]. It allows a flexible description of its processing pipeline, in particular the definition of arbitrary headers and packet parsers. P4 programs are compiled to so-called targets, e.g., the software switch BMv2 or switching ASICs. A compiled program offers the P4Runtime as an API so that P4 nodes can be re-configured by controllers during runtime.

B. P4 Pipeline

Figure 1 illustrates P4’s abstract forwarding model. A user-programmable parser reads an incoming packet and stores its header information in P4-internal header fields. They are carried with the packet through the P4 pipeline, possibly with additional metadata.

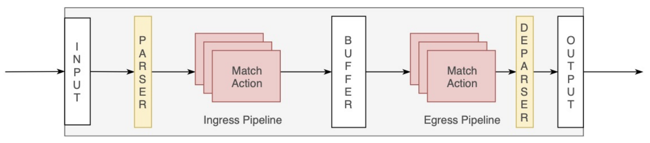


Fig. 1. P4 abstract forwarding model according to [8].

The P4 abstract forwarding model is divided into two stages, the ingress and the egress pipeline, which are separated by the packet buffer. For modularity, the ingress and egress pipeline can be further subdivided by control blocks (CB). Match+action tables (MATs) allow for packet-specific processing. They have entries consisting of match fields and match types that map packets to actions and parameters. One action may be defined to be carried out if no table entry matches a packet (table miss).

P4 offers in its core definition three match types: exact, lpm, and ternary. Exact implies that a packet header must contain the match field in the table entry, e.g. a given IPv4 address in the destination address field of an IP header. Lpm stands for longest prefix match which is well-known from standard IP forwarding. Ternary facilitates wildcard matches. A packet is processed at most once by the same MAT within the pipeline.

C. Registers

Information stored in metadata are only valid during a packets lifetime. To store information beyond the lifetime of a packet, P4 offers the ability to store information in so-called registers. Information stored in registers can be accessed during packet processing. We leverage registers to store data required for the protection mechanisms.

IV. In-Network Sensor Failover

In this section we first give an overview of the general context. Afterwards, we describe novel protection mechanisms for in-network sensor failover.

A. Overview

Figure 2 shows the concept of the proposed protection mechanisms. Two sensors periodically send data to an application over a network. To ensure reliability, both sensors send the same information, e.g., temperatures or velocity, but may send them with different periods.

An application may decide which data to use. The application might either use both data streams or only one data stream and, in case of an error, switch to the second data stream. However, this comes with an increase in program logic and developers have to deal with sensor failures. We propose to transfer the sensor failover to the network by leveraging programmable network devices, e.g. P4 switches. Two different modes of operation can be distinguished. By

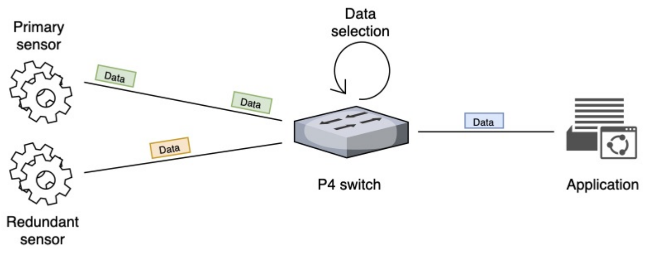
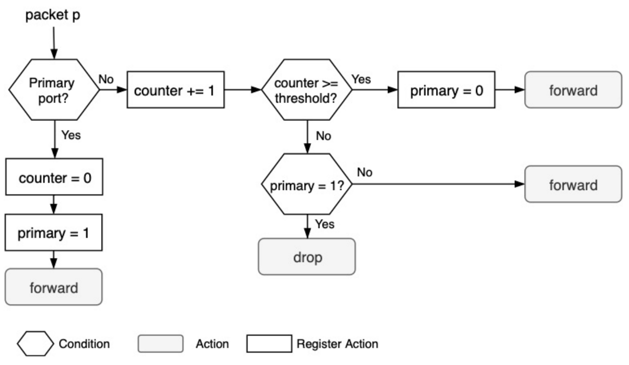
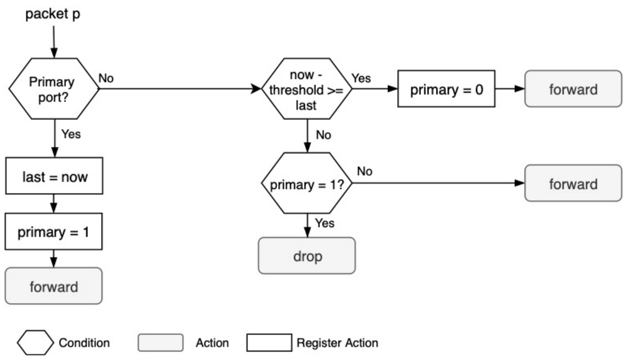


Fig. 2. Conceptual overview. Two redundant sensors provide information for a application.

default, the primary sensor data is forwarded to the application. With the aid of periodic messages from the redundant sensor, the P4 switch can detect if the primary sensor fails. If a failure is detected, data from the redundant sensor can be forwarded to the application.



(a) Counter-Based protection mechanism.



(b) Timer-Based protection mechanism.

Fig. 3. Overview ot the operations of the counter-based and timer-based protection mechanism.

B. Mechanisms

To detect the failure of the primary sensor, we leverage the time dependencies between the two data streams of the sensors. We propose two mechanisms to detect sensor failures, *counter-based* and *timer-based* failover, respectively. Figure 3(a) and Figure 3(b) illustrate the operations of the two protection mechanisms. We refer to actions involving register access as *Register Actions*, and actions without this access solely as *Action*.

1) Counter-Based Failover: The first protection mechanism is based on a counter approach. A counter is increased for each arriving data portion from the redundant sensor. In simplest form, the counter is increased by one and stored in a register field leveraging a *Register Action*. For each arriving data portion of the primary sensor, the counter is set to zero. If the counter exceeds a certain threshold T_c, the switch forwards the data from the redundant sensor to the application. The threshold has to be selected in such a way that the dependencies of the two data streams are taken into account. For example, if the redundant sensor transmits data twice as fast as the primary sensor, a failure of the primary sensor can be detected by a threshold of two and an increase by one. In such a case, at most one packet of the primary sensor is lost. If the periods of the two sensors are not multiples of each other, the same effect can be achieved by scaling the threshold and the respective increase of the counter.

If the primary sensor transmits faster than the secondary sensor, it cannot be guaranteed that at most one packet from the primary sensor will be lost. Furthermore, a change in the sensor periods may result in undesired behaviour, which is further described in Section VI. This protection mechanism is implemented for the high-performance P4 switching ASIC Tofino and demonstrated in Section VI.

2) Timer-Based Failover: The counter based approach is reliable if the sensor periods are stable. If the sensor periods change during operation, the relation between the intermediate arrival times and the configured threshold is no longer correct. In addition, the currently stored value in the counter is no longer valid. As a consequence, the counter must be reset. However, this may cause a delayed switch-over to the redundant sensor. To overcome this problem, we propose to use the actual intermediate arrival times for the protection mechanism instead of the counter-based relation among the arrival times. The *timer-based* approach utilizes packet timestamps which can be accessed during packet processing. For each arriving data portion of the primary sensor, the packet timestamp is saved in a special register. Data from the redundant sensor is only forwarded, if the elapsed time since the last data portion of the primary sensor exceeds a certain threshold T_t. This more advanced mechanism is implemented for the BMv2 software switch and can be accessed at Github[1]. The timer-based mechanism can also be implemented for the Tofino. For simplicity, we only implemented this mechanism for the BMv2. We give some examples of its advantages over the counter-based mechanism in Section VI.

V. Implementation

In this section we describe the P4 based implementation of the protection mechanisms presented in Section IV. First, we will give an overview of the P4 pipeline. Afterwards, we will

[1]Repository: https://github.com/uni-tue-kn/p4-source-protection

describe the important properties of the control block *Protect*, which implements the protection mechanisms. Finally, we give a rough overview of the control plane.

A. Overview

The implementation is based on a local Ethernet network and comprises local layer-2 switching and the applied protection mechanism. Figure 4 illustrates our implemented P4 pipeline.

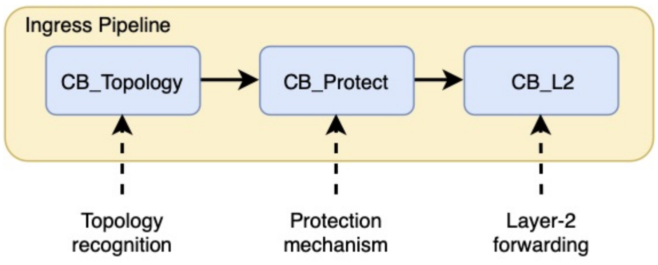


Fig. 4. Overview of the ingress pipeline.

The implementation solely requires the ingress part of the P4 pipeline. The P4 pipeline was introduced in Section III. The ingress pipeline is divided into three control blocks (CBs), named *CB_Topology*, *CB_Protect* and *CB_L2*. The control blocks *CB_Topology* and *CB_L2* are used for general network connectivity such as topology recognition and layer-2 forwarding. The protection mechanisms are implemented in the control block *CB_Protect*. Incoming packets traverse all three control blocks.

B. Control Block CB_Protect

The control block *CB_Protect* implements the two previously presented mechanisms. In order for the mechanisms to work, the switch must have access to several pieces of information. First and foremost, the switch must know the relation between the sensors and its physical interfaces, i.e. which interface corresponds to which sensor. Furthermore, the switch requires the configured threshold T_c or T_t. These information is dynamically provided with match+action tables (MATs). During packet processing, the information is made available by matching on these MATs and storing the required information in metadata fields. As soon as the period of the sensors changes, the contents of the MATs can be updated by the control plane. As a consequence, we can react dynamically to the changes. In addition to the interface and threshold information, the last timestamp of the primary sensor and the counter must be stored. We leverage registers that are available in the software switch BMv2 as well as in the Tofino.

C. Control Plane

The control plane is responsible for filling the different MATs with entries. To that end, it provides an interface for runtime changes and updates the MATs accordingly. It utilizes information of a proprietary topology detection mechanism to calculate the appropriate forwarding rules of the network to enable local layer-2 forwarding.

VI. Evaluation & Discussion

In this section we illustrate the functionality and effectiveness of the two introduced protection mechanisms. To accomplish this, we perform experiments in our testbed using our prototype. We first explain the general setup of our experiments. Afterwards, we introduce our evaluation metrics. Finally, we describe the experimental results and explain some theoretical examples.

A. Methodology

1) General Setup: The hardware testbed consists of three servers physically connected to a Tofino Edgecore Wedge 100BF-32X as shown in Figure 2. The servers are based on an Intel Xeon Scalable Gold 6134 (8x 3.2 GHz) and 4x 32 GB RAM. The Tofino Edgecore Wedge 100BF-32X is a high-performance P4 switch with 32 100G ports. Two servers thereby act as sensors, the third server mimics an application. Both sensors send data to the application with different periods p_0 and p_1.

2) Metric: We evaluate the arrival of the packets of the two sensors at the application. During operation we simulate a sensor failure by disconnecting the link between the primary sensor and the P4 switch. We demonstrate that the correct configuration of the mechanism is essential and show by a theoretical example that the *timer-based* approach is superior to the *counter-based* approach.

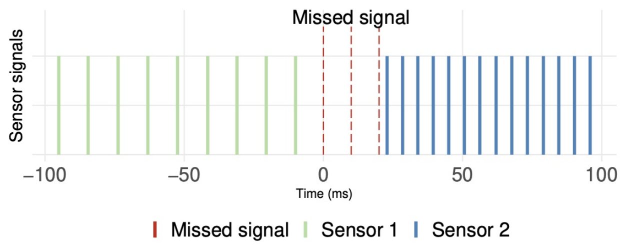
B. Counter-Based Protection

To illustrate the influence of the threshold on the counter-based mechanism, we consider the experiment as described in Section VI-A. Figure 5(a) reflects the result of an incorrect configuration of the *counter-based* mechanism. The primary sensor sends with a period of $p_1 = 10$ ms and the redundant sensor with a period of $p_2 = 5$ ms. Since the secondary sensor transmits twice as fast as the primary sensor, a failure of the primary sensor can be detected by a threshold of $T_c = 2$. In this experiment the threshold was falsely set to $T_c = 5$. Figure 5(a) shows the incoming data packets from the primary sensor (sensor 1) and the redundant sensor (sensor 2) at the application for this configuration.

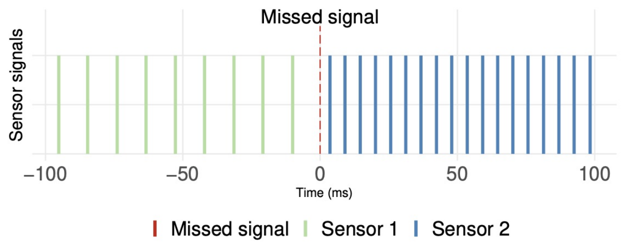
At time $t = 0$, the first packet of the primary sensor is lost. Subsequently, due to the wrong threshold, two additional packets of the primary sensor are lost before the system switches to the redundant sensor. In contrast, Figure 5(b) shows that with a properly tuned threshold, only one packet of the primary sensor is lost.

C. Timer-based Protection

With constant sensor periods, the *timer-based* mechanism performs as well as the *counter-based* mechanism. However, as soon as the periods change during operation, the *timer-based* approach is superior. Two strategies can be pursued for the *counter-based* approach. After a period change, the

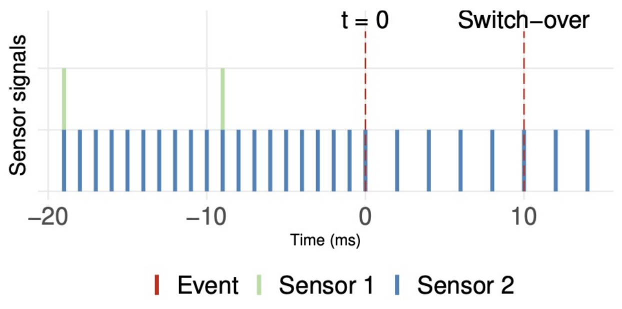


(a) $p_1 = 10$ ms, $p_2 = 5$ ms, $T_c = 5$.

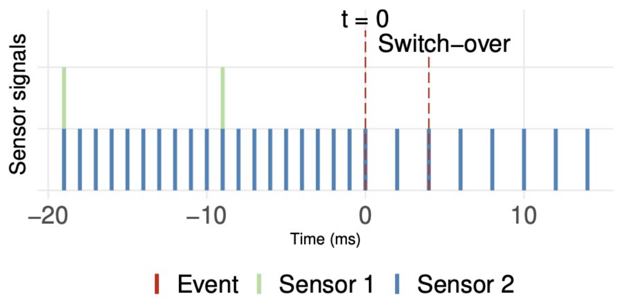


(b) $p_1 = 10$ ms, $p_2 = 5$ ms, $T_c = 2$.

Fig. 5. Influence of threshold on sensor failover for the counter-based protection mechanism.

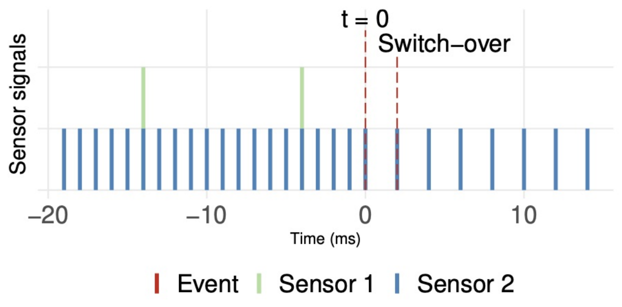


(a) Counter-based mechanism with reset.

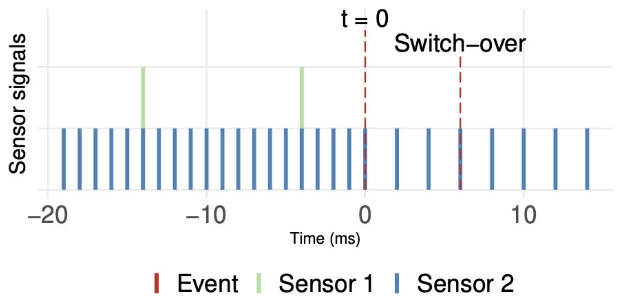


(b) Timer-based mechanism.

Fig. 6. Different behaviour of the two mechanisms during a period change. The *counter-based* mechanism requires more time for the switch-over than the *timer-based* mechanism.



(a) Counter-based mechanism without reset.



(b) Timer-based mechanism.

Fig. 7. Different behaviour of the two mechanisms during a period change. The *counter-based* mechanism erroneously forwards data from the secondary sensor.

counter can either be reset or maintained. We now consider the case that the counter is reset. Figure 6(a) illustrates an example. Note that all transmitted signals are displayed. We further assume that at $t = 0$ the primary sensor fails and that the counter is reset. At the same time the period of the redundant sensor changes from 1 ms to 2 ms. The *counter-based* approach forwards the data of the redundant sensor at time $t = 10$ ms, as all information before the period change is lost. In contrast to the *counter-based* mechanism, the timer based mechanism performs the switch-over as intended at $t = 4$ ms. Figure 6(b) visualizes the differences.

If the counter is maintained at a period change, the *counter-based* mechanism will still not behave correctly. Lets assume similar settings as in the previous example. The primary sensor transmits data with period $p_1 = 10$ ms, the secondary sensor with $p_2 = 1$ ms. At time $t = 0$ the primary sensor fails, the counter equals four and the period of the secondary sensor switches again to $p_2 = 2$ ms. Figure 7(a) illustrates this setup. As the counter has not been reset, the switch erroneously switches to the redundant sensor at time $t = 2$. Again, Figure 7(b) shows that the *timer-based* mechanism is not affected by this problem.

VII. Conclusion

Sensors in critical systems must provide applications with redundant information. Applications have to decide how to handle the different data streams and how to react to the failure of one of the data streams. This is not only prone to errors but also leads to duplication in application logic. In this paper we proposed two mechanisms to move the detection of sensor errors to the network to make the redundant data transmission transparent for the application. We have shown the disadvantages of a simple counter-based mechanism and presented a more complex timer-based mechanism for the BMv2.

REFERENCES

[1] A. Atlas and A. Zinin, "RFC5286: Basic Specification for IP Fast Reroute: Loop-Free Alternates ," Sep. 2008.

[2] "ITU-T Recommendation G.7712/Y.1703 (2010), Internet protocol aspects – Operation, administration and maintenance," ITU, Sep. 2010.

[3] M. F. Al-Naday, M. J. Reed, D. Trossen, and K. Yang, "Information resilience: source recovery in an information-centric network," *IEEE Network*, vol. 28, no. 3, pp. 36–42, 2014.

[4] M. Hoefling, C. G. Mills, and M. Menth, "Distributed load balancing for the resilient publish/subscribe overlay in sedax," *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 147–160, 2017.

[5] M. Ryll and S. Ratchev, "Towards a publish / subscribe control architecture for precision assembly with the data distribution service," in *Micro-Assembly Technologies and Applications*, S. Ratchev and S. Koelemeijer, Eds. Boston, MA: Springer US, 2008, pp. 359–369.

[6] J. Campelo, F. Rodriguez, A. Rubio, R. Ors, P. Gil, L. Lemus, J. Busquets, J. Albaladejo, and J. Serrano, "Distributed industrial control systems: a fault-tolerant architecture," *Microprocessors and Microsystems*, vol. 23, no. 2, pp. 103 – 112, 1999.

[7] B. Rostamzadeh, H. Lonn, R. Snedsbol, and J. Torin, "Dacapo: a distributed computer architecture for safety-critical control applications," in *Proceedings of the Intelligent Vehicles '95. Symposium*, 1995.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, 2014.

*

2.2 Comparison of Fast-Reroute Mechanisms for BIER-Based IP Multicast

Comparison of Fast-Reroute Mechanisms for BIER-Based IP Multicast

Daniel Merling, Steffen Lindner, Michael Menth
Chair of Communication Networks, University of Tuebingen, Tuebingen, Germany
{daniel.merling, steffen.lindner, menth}@uni-tuebingen.de

***Abstract*—IP multicast (IPMC) delivers one-to-many traffic along distribution trees. To that end, conventional IPMC requires state in forwarding devices for each IPMC group. This limits scalability of IPMC because forwarding state in core devices may be extensive and updates are necessary when IPMC groups or the topology change. The IETF introduced Bit Index Explicit Replication (BIER) for efficient transport of IPMC traffic. BIER leverages a BIER header and IPMC-group-independent forwarding tables for forwarding of IPMC packets in a BIER domain. However, legacy devices do not support BIER. In contrary, two SDN-based implementations for OpenFlow an P4 have been published recently. In this paper, we assess BIER forwarding which may be affected by network failures. So far there is no standardized procedure to handle such situations. Two concepts have been proposed. The first approach is based on Loop-Free Alternates. It reroutes traffic to suitable neighbors in the BIER domain to steer traffic around the failure. The second approach is a tunnel-based mechanism that tunnels BIER packets to appropriate downstream nodes within the BIER distribution tree. We explain and compare both approaches, and discuss their advantages and disadvantages.**

***Index Terms*—Software-Defined Networking, Bit Index Explicit Replication, Multicast, Resilience, Scalability**

I. Introduction

IP multicast (IPMC) is used for services like IPTV, commercial stock exchange, multicast VPN, content-delivery networks, or distribution of broadcast data. Figure 1 shows the concept of IPMC.

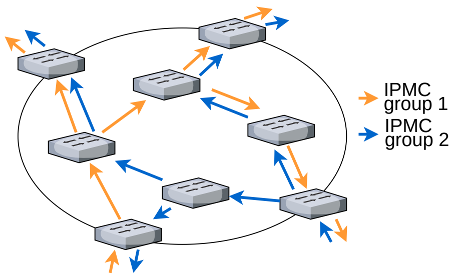


Figure 1: Two multicast distribution trees.

IPMC efficiently distributes one-to-many traffic by replicating packets and forwarding only one packet per link. Hosts join an IPMC group to receive the traffic addressed to that group. Forwarding devices maintain IPMC-group-dependent state to forward packets to the right neighbors. This decreases the scalability of IPMC for the following reasons. First, a large number of IMPC groups require a significant amount of forwarding state in core devices. Second, when subscribers of an IPMC group change, i.e., devices join or leave the group, the forwarding state needs to be updated. Third, when the topology changes or in case of a failure, the forwarding information base of possibly many devices has to be adapted.

The IETF presented BIER [1] as an efficient transport mechanism for IPMC traffic. BIER introduces a BIER domain, where only ingress routers maintain IPMC-group-dependent state. Ingress routers of the BIER domain encapsulate IPMC packets with a so-called BIER header which contains the destinations of the packet. Within the BIER domain, BIER packets are forwarded along distribution trees from the source to the destinations. Thereby only a single packet is transmitted per link. Finally, egress nodes remove the BIER header. Forwarding in the BIER domain is based on two components. First, the BIER header which contains a bit string that identifies receivers of a packet within the BIER domain. Second, the so-called Bit Index Forwarding Table (BIFT) which is the routing table of BIER devices. The entries of the BIFT are derived from information from the routing underlay, e.g., the Interior Gateway Protocol (IGP).

When a primary next-hop (NH) is unreachable due to a failure, an entire set of downstream destination nodes does not receive the traffic. When a failure is detected, IGP converges, new distribution trees are calculated, and the BIFTs are updated. This process requires a significant amount time. Therefore, BIER would benefit greatly from a fast protection mechanism that delivers traffic in the meantime. For unicast, several fast reroute (FRR) mechanisms [2] have been proposed which protect against the failure of single links or nodes until the forwarding information base is updated. FRR mechanisms use pre-computed backup entries to quickly reroute traffic when the primary NH is unreachable. No signaling between devices is necessary. Two FRR concepts for BIER have been proposed. First, LFA-based BIER-FRR [3] leverages a FRR mechanism called Loop-Free Alternates (LFAs) [4] that has been initially proposed for IP unicast. Failures are bypassed by forwarding traffic to alternative BIER NHs. Second, tunnel-based BIER-FRR tunnels traffic through the routing underlay, leveraging its FRR capabilities to steer traffic around the failure. We proposed this mechanism at the IETF [5].

However, legacy devices do not support BIER. On the contrary, the flexibility of SDN-based technologies have been leveraged recently to successfully implement BIER with OpenFlow [6] and in P4[1]. This allows the deployment of BIER

The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/1-2. The authors alone are responsible for the content of the paper.

[1]https://github.com/uni-tue-kn/p4-bier

and facilitates the implementation of additional BIER-related features, e.g. BIER-FRR.

In this paper we review LFA-based and tunnel-based BIER-FRR. First, we propose changes to tunnel-based BIER-FRR to reduce the number of forwarding entries. Then, we point out major shortcomings of the LFA-based approach and present extensions to resolve the issues. Further, we compare both mechanisms by discussing their protection capabilities, and overhead in terms of header size and forwarding state.

The paper is structured as follows. Section II describes related work for conventional and SDN-based multicast, and BIER. We review BIER in Section III. Section IV gives a primer on LFAs. Then, in Section V we explain tunnel-based BIER-FRR. Afterwards, we describe LFA-based BIER-FRR in Section VI, and point out its shortcomings and propose extensions in Section VII. Finally, we compare and discuss both approaches in Section VIII. We conclude the paper in Section IX.

II. Related Work

In this section we first discuss related work for conventional and SDN-based multicast. Afterwards, we review related work for BIER.

A. *Multicast*

In [7] the authors provide an overview of the early development of multicast. The authors of [8] discuss the limited scalability of conventional IP multicast in terms of the number of forwarding entries. They propose an extension to the multicast routing protocol MOSPF to reduce the number of required forwarding entries. Li et al. [9] propose an architecture to partition the multicast address space to increase scalability of IP multicast in data center topologies.

B. *SDN-Based Multicast*

The surveys [10], [11] provide a detailed overview of SDN-based multicast. We discuss only some of the mentioned papers. The authors of [12] introduce software-defined multicast (SDM), an OpenFlow-based approach that aims at providing a well-managed multicast platform for over-the-top and overlay-based live streaming services. SDM is specifically engineered for the needs of P2P-based video stream delivery. They further develop their idea of SDM in [13] by adding support for fine-granular traffic engineering capabilities. Lin et al. [14] present a multicast model to construct so-called multi-group shared trees. By deploying distribution trees that cover multiple multicast groups simultaneously, the entire network is covered with a small number of trees.

C. *Protection of SDN-Based Multicast*

Kotani et al. [15] propose to leverage multiple simultaneously deployed multicast trees for protection. An ID in the packet header determines along which distribution tree a packet is forwarded. When a tree is affected by a failure, the controller reconfigures the senders to forward traffic on a backup tree. The authors of [16] follow a similar approach where they leverage primary and backup trees identified by a VLAN tag. When a switch detects a failure, it reroutes the packets on a working backup tree that contains all downstream nodes. This is accomplished by switching the VLAN tag in the packet header.

D. *BIER Related Work*

Giorgetti et al. [6], [17] provide an implementation for both, conventional IPMC and BIER forwarding in OpenFlow. They leverage MPLS headers to encode the BIER bit string, which limits the bit string length, and thereby the number of destinations, to a maximum of 20. However, a local BIER agent is required to run on the switches to support arbitrary destinations. BIER-TE [18] extends BIER with traffic engineering capabilities. BIER-TE leverages the same header format as BIER and supports explicit coding of a distribution tree in the BIER header. However, BIER and BIER-TE are not compatible. The authors of [19] present a P4-based implementation of BIER and BIER-TE and present different demo scenarios to show the feasibility and the advantages of BIER(-TE). The authors of [20] propose 1+1 protection for BIER-TE. Traffic for each IPMC group is forwarded on two disjoint distribution trees simultaneously. The trees share as few network components as possible to still deliver traffic when one tree is interrupted by a failure. However, the approach requires two forwarding planes, and in the failure free case twice the amount of network resources are occupied.

III. Bit Index Explicit Replication (BIER)

The following section reviews BIER [1]. First, we describe its concept, the structure of the Bit Index Forwarding Table (BIFT), the BIER forwarding procedure, and a forwarding example. Afterwards we explain a compact representation of the BIFT, and characteristics of the BIER topology.

A. *BIER Concept*

BIER is based on a layered architecture, consisting of routing underlay, BIER layer, and IPMC layer. Figure 2 illustrates the relation between these components.

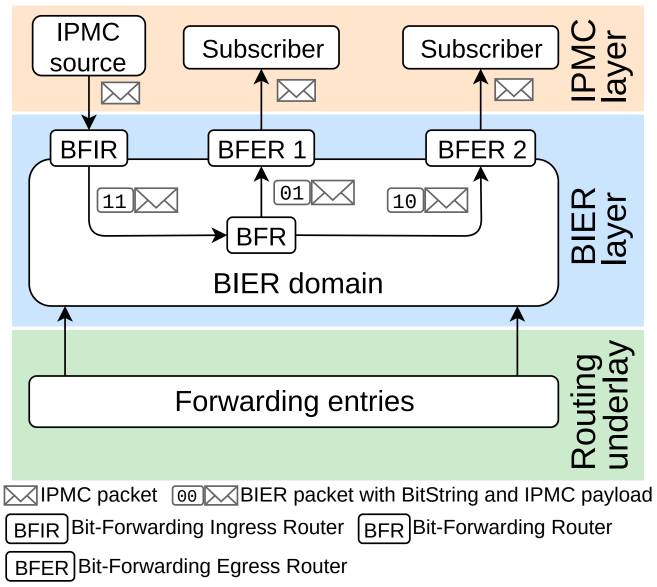

IPMC source
Subscriber
Subscriber
IPMC layer
BFIR
BFER 1
BFER 2
11
01
10
BFR
BIER domain
BIER layer
Forwarding entries
Routing underlay
IPMC packet
00
BIER packet with BitString and IPMC payload
BFIR Bit-Forwarding Ingress Router
BFR Bit-Forwarding Router
BFER Bit-Forwarding Egress Router

Figure 2: Layered architecture of BIER; it shows the relation between routing underlay, BIER layer, and IPMC layer.

The BIER layer serves as a point-to-multipoint tunnel for IPMC traffic through a BIER domain. The BIER domain consists of bit forwarding ingress routers (BFIRs), bit forwarding routers (BFRs), and bit forwarding egress routers (BFERs). A BIER-capable device can be BFIR, BFR and BFER at the same time. When an IPMC packet enters the domain, the BFIR pushes a BIER header onto the IPMC packet. The BIER header identifies all receivers (BFERs) of the packet within the BIER domain. To that end, it contains a bit string which has to be at least as long as the number of BFERs in the BIER domain. In the following, 'BitString' refers to the bit string in the BIER header of the packet. Each BFER is assigned to a bit position in the BitString, starting with the least-significant bit. An activated bit means that the corresponding BFER must receive a copy of the BIER packet. BFRs forward BIER packets according to theor BitString along distribution trees to multiple BFERs.

Paths in the BIER domain are derived from the routing underlay, e.g., the IGP. As a consequence, BIER traffic follows the same paths as the corresponding unicast traffic from source to destination. At the domain boundary, BFERs remove the BIER header and pass the IPMC packet to the IPMC layer.

B. BIFT Structure

Table 1 shows the BIFT of BFR 1 from Figure 3. For each BFER, the BIFT contains one forwarding entry that consists of the primary NH and the so-called Forwarding Bit Mask (F-BM). The F-BM is a NH-specific bit string similar to the bit string in the BIER header. It indicates the BFERs with the same NH. In one particular F-BM, only bits of BFERs that are reached over the same NH are activated. During forwarding, BFRs use the F-BM to clear bits from the BitString.

C. BIER Forwarding

When a BFR receives a BIER packet, it stores its BitString to account to which BFERs the packet needs to be sent. We refer to that stored bit string by the term 'remaining bits'. The following procedure is repeated until the remaining bits do not contain any activated bits anymore.

The BFR determines the least-significant activated bit in the remaining bits. This bit indicates the BFER to be processed. Then, the BFR performs a looks up in the BIFT to get the NH and F-BM for that BFER. After a successful match, the BFR creates a copy of the received BIER packet. The BFR clears the BFERs from the BitString of the packet copy that have a different NH. To that end, the BFR performs a bitwise AND operation of the F-BM and the BitString of the packet copy. Then the BFR writes the result into the BitString of the packet copy. This procedure is called applying the F-BM. Thus, only bits that correspond to BFERs which share the same primary NH remain active in the BitString of the packet copy. Clearing other bits avoids duplicates at the receivers. Afterwards, the packet copy is forwarded to the NH. Finally, the BFERs, to which a packet has just been sent, are removed from the remaining bits. To that end, a bitwise AND operation of the bitwise complement of the F-BM and the remaining bits is performed.

D. BIER Forwarding Example

Figure 3 shows an example topology with four BFRs. Each BFR is in addition a BFIR and a BFER. Table 1 shows the BIFT of BFR 1.

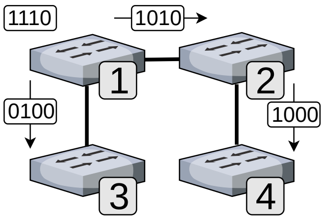


Figure 3: BIER topology and BitStrings of forwarded BIER packets.

BFER	NH	F-BM
1	-	-
2	2	1010
3	3	0100
4	2	1010

Table 1: BIFT of BFR 1.

BFR 1 receives a BIER packet with the BitString `1110`. The least-significant activated bit in the remainings bits identifies BFR 2. Therefore, BFR 1 creates a copy of the packet, applies the corresponding F-BM `1010`, and forwards the packet copy with the BitString `1010` to BFR 2. This sends a packet to BFER 2 and BFER 4. Afterwards, the bits of the F-BM are cleared from the remaining bits `0100`. The least-significant activated bit in the remaining bits corresponds to BFER 3. The F-BM is applied and a packet clone with the BitString `0100` is forwarded to the NH which is BFR 3. After clearing the F-BM from the remaining bits, processing stops because no active bits remain.

E. Compact BIFT

The number of entries of the BIFT scales with the number of BFERs. For improved scalability in terms of forwarding entries, the authors of [21] propose a compact representation of the BIFT that requires only one forwarding entry per neighbor. To that end, all entries with the same NH and F-BM are aggregated. As a result, all BFERs indicated in the F-BM share a single forwarding entry. During lookup, an entry is considered a match when at least one of the associated BFERs is a destination of the BIER packet. Table 2 shows the compact BIFT of BFR 1 from Figure 3.

BFERs	NH	F-BM
2, 4	2	1010
3	3	0100

Table 2: Compact BIFT of BFR 1.

F. Characteristics of the BIER Topology

In this paragraph we first discuss the impact of differences between the Layer 3 topology and BIER topology. Afterwards, we review how BIER devices detect whether BIER neighbors are still reachable.

1) Differences Between Layer 3 Topology and BIER Topology: In a Layer 3 topology some Layer 3 devices may not be BIER capable. Thus, the BIER topology may be different from the Layer 3 topology. Neighbors in the BIER topology are either connected directly to each other, or through at least one intermediate Layer 3 device that is no BIER device. BIER nodes receive information about their connection to their neighbors from the routing underlay. If two BIER neighbors are directly adjacent, they forward packets over Layer 2 to each other. If they are not directly adjacent, the BIER neighbors leverage a Layer 3 tunnel to exchange packets. In both cases forwarding still follows the paths from the routing underlay.

2) Detection of Unreachable NHs: To quickly detect unreachable BIER neighbors, the authors of [22] propose bidirectional forwarding detection (BFD) [23] for BIER. When a BFD is established between two BIER nodes, they periodically exchange notifications to observe the reachability.

IV. Loop-Free Alternates

In this section we explain the concept of Loop-Free Alternates (LFAs) [4]. Afterwards, we review extensions for improved protection capabilities and loop detection.

A. Foundations of LFAs

LFAs implement a FRR mechanism for IP unicast traffic that prevents rerouting loops. Figure 4 shows the concept of LFAs.

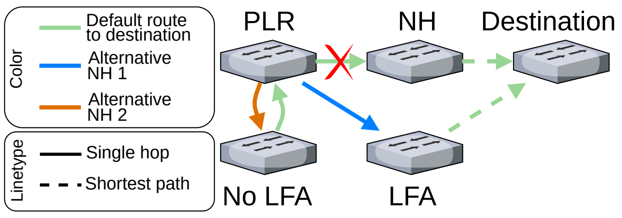


Figure 4: Concept of LFAs.

When a node cannot reach a primary NH, it acts as point of local repair (PLR), i.e., it leverages a pre-computed backup entry to reroute the packet via an alternative NH on a backup path towards the destination. Such neighbors are called LFAs and they have to be chosen in a way that rerouting loops are avoided. Some neighbors must not be chosen as LFAs because rerouting the packet would result in a forwarding loop.

LFAs have different properties for protection and loop avoidance. Some protect against link failures, others against node failures. Link-protecting LFAs (LP-LFAs) have a shortest path towards the destination that does not include the link between PLR and primary NH. Thus, LP-LFAs protect against the failure of the link between PLR and primary NH. The authors of [24] and [25] analyze the protection capabilities of LP-LFAs with a comprehensive set of topologies. They find that LP-LFAs protect only 70% of destinations against single link failures. Furthermore, LP-LFAs may cause loops when at least one node or multiple links fail instead of a single link only. To protect against the failure of the primary NH, node-protecting LFAs (NP-LFAs) have a shortest path to the destination that does not include the primary NH. In [24] the authors evaluate NP-LFAs in different scenarios on a large set of topologies. They show that NP-LFAs prevent loops for single link and single node failures, but they protect only 40% of destinations against single link failures.

B. Extensions for LFAs

In this paragraph we explain remote LFAs (rLFAs), topology independent LFAs (TI-LFAs), and explicit LFAs (eLFAs) to complement LFAs for increased protection capabilities. All three LFA variants support link and node protection. We indicate the protection mode with the prefix ’LP-’ for link protection, and ’NP-’ for node protection. Furthermore, we review a loop detection mechanism for LFAs. Figure 5 shows the concept of rLFAs, TI-LFAs, and eLFAs, which we explain in detail in the following.

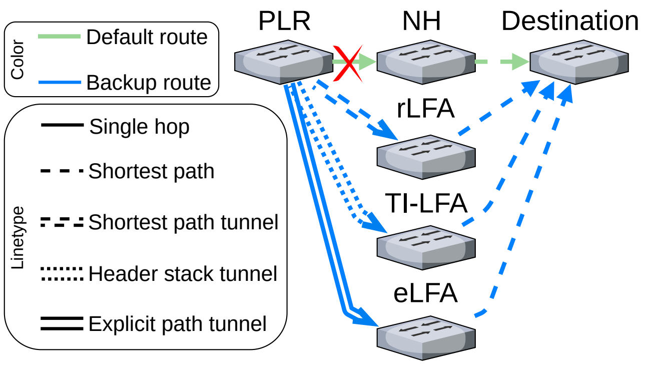


Figure 5: Concept of rLFAs, TI-LFAs, and eLFAs.

1) Remote LFAs (rLFAs): rLFAs [26] are remote nodes in the network. When the PLR cannot reach a primary NH, the packet is rerouted through a shortest path tunnel to the rLFA. From there, the packet is forwarded on a shortest path towards the destination. In [26] the authors prove that there is always a LP-rLFA to protect against a single link failure in unit-link-cost topologies. However, the authors of [25] find that this property does not hold for topologies with arbitrary link costs. NP-rLFAs cannot protect against all single link or single node failures.

2) Topology-Independent LFAs (TI-LFAs): TI-LFAs [27] are remote nodes in the network. When the primary NH is unreachable, the PLR leverages a header stack of IP headers to deviate traffic to the TI-LFA. The TI-LFA then sends the original packet on a shortest path towards the destination. As long as there is still a working shortest path to the destination, LP-TI-LFAs can protect against any single link failure, and NP-TI-LFAs against any single node failure.

3) Explicit LFAs (eLFAs): eLFAs [25] follow a similar concept as TI-LFAs. An eLFA is a remote node that serves as tunnel-end point when the PLR cannot reach the primary NH. The PLR reroutes the packet through an tunnel on an explicit path to the eLFA. The eLFA then forwards the packet on a shortest path to the destination. In contrast to TI-LFAs, eLFAs leverage additional forwarding entries for explicit paths to prevent an IP header stack. The authors of [25] evaluate eLFAs on a comprehensive set of different topologies. As long as the destinaton is still reachable, LP-eLFAs protect against any single link failure and NP-eLFAs protect against any single node failure.

4) Loop Detection: LFAs and all of its variants share the shortcoming that their deployment may cause forwarding loops [24], [25] in case of unprotected failures. In [24] the authors present a loop detection mechanism for LFAs. It is based on a bit string in the packet header where each forwarding device in the network is assigned a bit position. When a node needs to reroute a packet, it checks whether its own bit is activated. If this is not the case, the node activates the bit and reroutes the packet. However, if the bit is already activated, the packet has been rerouted by the node before, and thus, the packet is dropped to prevent a loop. In [25] the authors describe loop detection for all LFA variants.

V. Tunnel-Based BIER-FRR

In this section we review tunnel-based BIER-FRR. We introduced this mechanism at the IETF [5]. First, we describe the concept, explain two modes of operation and an example. Then, we present changes to tunnel-based BIER-FRR for deployment with the compact BIFT. Finally, we discuss forwarding state.

A. Concept

When a BFR cannot forward a packet to a NH, the neighbor may still be reachable on a backup path. Tunnel-based BIER-FRR tunnels traffic through the routing underlay around the failure to BIER nodes downstream in the BIER distribution tree. A tunnel may be affected by the same failure but the routing underlay quickly restores connectivity with FRR mechanisms. With link protection, tunnel-based BIER-FRR tunnels the BIER packet to the NH. With node protection, BIER packets with adjusted BitStrings are tunneled to the next-next-hops (NNHs). Additionally, one BIER packet is tunneled to the NH to deliver a packet if only the link between PLR and NH failed.

Protection capabilities of tunnel-based BIER-FRR depend on the properties of the routing underlay. Tunnel-based BIER-FRR protects against any single component failure which can be handled by FRR mechanisms in the routing underlay. We describe the operation of tunnel-based BIER-FRR for link and node protection based on the normal BIFT.

1) Link Protection: Tunnel-based BIER-FRR with link protection does not require changes to the BIFT. When a primary NH is unreachable, the packet copy is tunneled to the NH instead of being forwarded on Layer 2. The routing underlay leverages IP-FRR to deliver the packet to the NH.

2) Node Protection: Tunnel-based BIER-FRR with node protection tunnels BIER packets to the NNHs. However, usually the NH adapts the BitString before the packet is forwarded to the NNH. Thus, before the packet is tunneled, the PLR performs modifications on the BitString that are usually done by the NH, i.e., applying the F-BM. To that end, backup entries in the BIFT are required which consist of a backup NH, and a backup F-BM. There are two categories of backup entries. First, for BFERs that are also NHs. In such backup entries, the NH is the backup NH and in the backup F-BM only the bit of the BFER is activated. This tunnels a packet to the NH in case only the link between PLR and NH failed. The second category of backup entries is for BFERs that are not NHs. For their entries, the backup NH is the NNH towards the BFER. The backup F-BM is the primary F-BM of the NH for the NNH.

When a primary NH is unreachable, the BFR performs three operations. First, the BFR applies the primary and the backup F-BM to the packet clone. The primary F-BM clears BFERs from the BitString that have a different NH. The backup F-BM clears BFERs from the BitString that have a different NNH. This leaves only bits of BFERs active that are activated in both, the primary and backup F-BM, i.e., all BFERs that have the same NH and the same NNH. Second, the packet copy is tunneled to the backup NH. Third, only bits that are active in both, the primary and backup F-BM are cleared from the remaining bits.

B. Forwarding Example

Figure 6 shows a BIER topology with a node failure where each BFR is also a BFIR and BFER. Table 3 displays the BIFT of BFR 1 with backup entries for node protection.

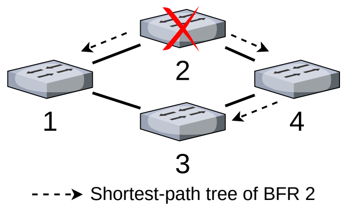


Figure 6: BIER topology with a node failure. The shortest-path tree of BFR 2 is shown to derive the backup F-BM of BFR 1 for BFER 4.

BFER	NH	F-BM
2	2	1010
	2	0010
3	3	0100
	3	0100
4	2	1010
	4	1100

Table 3: BIFT of BFR 1 with backup entries for node protection.

BFR 1 processes a packet with the BitString `1000`. The least-significant activated bit identifies BFER 4. However, the primary NH BFR 2 is unreachable. Thus, both, the primary F-BM `1010` and the backup F-BM `1100` are applied to the BitString of the packet copy. This leaves the BitString `1000` and the packet is tunneled to BFR 4 through the routing underlay. Bits that are activated in both, the primary and backup F-BM are cleared from the remaining bits which leaves `0000` and processing stops. The packet is eventually delivered by the routing underlay to BFR 4.

C. Compact BIFT

When the compact BIFT is used, tunnel-based BIER-FRR with link protection can be deployed as described in Section V-A1. Tunnel-based BIER-FRR with node protection requires two modifications. First, multiple backup entries are required for each primary forwarding entry. In the compact BIFT, each primary forwarding entry corresponds to one specific NH. For each NNH of the NH, one backup entry is required. The backup entries are calculated as described in Section V-A2. Second, when a BFR detects that a specific NH is unreachable, it matches incoming packets on the backup entries of the affected primary entry instead.

D. State Discussion

Tunnel-based BIER-FRR requires one backup entry for each primary entry. Therefore, in a topology with n BFERs the normal BIFT with backup entries contains $n + n$ forwarding entries. Deployment with the compact BIFT requires significantly fewer forwarding entries because the average number of neighbors is significantly smaller than the number of destinations in a network. In a topology with an average node-degree of k, each node has k neighbors, and each NH has $k-1$ NHs on average. As a result the average number of forwarding entries per node is the sum of primary forwarding entries and backup entries $k + k \cdot (k-1)$.

VI. LFA-Based BIER-FRR

In this section we review LFA-based BIER-FRR [3]. We explain the concept, derivation of backup entries, and a forwarding example.

A. Concept

LFA-based BIER-FRR leverages backup entries in the BIFT to deviate traffic on backup paths when the primary path is interrupted. A backup entry consists of a backup NH, and a backup F-BM. When a primary NH is unreachable, further processing depends on the availability of a backup entry. If there is no backup entry, the bit of the BFER is cleared from the remaining bits and no packet is delivered to this particular BFER. Processing resumes with the next BFER. If there is a backup entry, further packet processing differs in three ways from regular BIER forwarding. First, the PLR applies the backup F-BM instead of the primary F-BM to the BitString of the packet clone. Second, the BIER packet is forwarded to the backup NH instead of the primary NH. Third, the bits of the backup F-BM instead of the primary F-BM are cleared from the remaining bits. Afterwards, the next BFER is processed.

B. Derivation of Backup Entries

We describe how we derive a backup entry consisting of a backup NH and a backup F-BM for a specific primary entry. First, we identify BIER neighbors that are LFAs as described in Section IV. LFA computation has to be performed on the BIER topology because Layer 3 LFAs may not be available on BIER layer due to topology differences. If no LFA is available, the primary forwarding entry remains without a backup entry. If there is an LFA L, it is selected as the backup NH. The activated bits in the backup F-BM are determined as follows. The bit that corresponds to an arbitrary BFER B is activated in the backup F-BM only if one of the two following conditions is fulfilled. First, L is an LFA to protect B. Second, L is the primary NH on the path to B. This aggregates all BFERs that are reached on a primary or backup path where L is the NH.

C. Forwarding Example

Figure 7 shows a BIER topology with a failed link between BFR 1 and 2. Each BFR is both a BFIR and a BFER. Table 4 contains the BIFT of BFR 1 with backup entries for link protection.

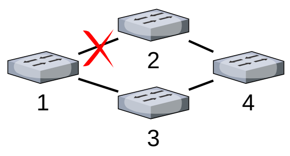


Figure 7: BIER topology with a link failure.

BFER	NH	F-BM
2	2	1010
	-	-
3	3	0100
	-	-
4	2	1010
	3	1100

Table 4: BIFT of BFR 1 with backup entries for link protection.

BFR 1 processes a BIER packet with the BitString `1110`. The least-significant activated bit identifies BFER 2. However, the primary NH BFR 2 is unreachable and there is no backup entry. Thus, the bit for BFER 2 is cleared from the remaining bits `1100` and no packet is sent. The next destination is BFER 3. Since the primary NH BFR 3 is reachable, the primary F-BM is applied and a packet clone with the BitString `0100` is forwarded to BFR 3. Clearing the F-BM from the remaining bits leaves only one bit activated `1000` which corresponds to BFER 4. However, the primary NH BFR 2 is unreachable. Thus, the backup F-BM is applied and a packet copy with the BitString `1000` is forwarded to the backup NH BFR 3. After the backup F-BM has been cleared from the remaining bits, no activated bits remain and processing stops. BFR 3 then forwards the packet to its destination BFR 4.

VII. Extensions for LFA-Based BIER-FRR

In this section, we expose major shortcomings of LFA-based BIER-FRR in terms of matching order, coverage, and forwarding state, and propose solutions. In the end we discuss scalability in terms of forwarding entries.

A. Matching Order

In the previous example two packets are forwarded to BFR 3. This is caused by the order in which receivers of a packet are processed. The following scenario describes when more than one packet is forwarded to one specific NH P. First, a packet is forwarded to the primary NH P towards a set of BFERs. Second, another BFER that should receive the packet is processed but its primary NH is unreachable. However, P is the backup NH. Thus, a second packet is forwarded to P on a backup path. To avoid sending multiple packets over one link, it is necessary to first process forwarding entries whose primary NH is unreachable. Then, no additional packet is sent because the backup F-BM aggregates primary and backup paths that have the same NH.

B. Coverage

Depending on the topology, LFAs cannot protect against arbitrary single component failures. rLFAs protect against any single link failure on unit-link-cost topologies. TI-LFAs and eLFAs guarantee protection against any single component failure on arbitrary topologies. However, the deployment of each of the three LFA extensions requires some sort of IP or segment routing tunnel. Nevertheless, full protection is an important property and we suggest to augment LFA-based BIER-FRR with rLFAs, TI-LFAs, or eLFAs to increase the coverage. rLFAs, TI-LFAs, and eLFAs need to be BFRs. Therefore, computations have to be performed on the BIER topology because not all Layer 3 devices may be BIER devices.

C. Compact BIFT

We explain scalability issues of LFA-based BIER-FRR and propose a solution that requires changes to how backup entries are derived.

1) Problem Statement and Solution: LFA-based BIER-FRR has been described for the BIFT that contains one primary forwarding entry per BFER. In its proposed form LFA-based BIER-FRR is incompatible with the compact representation of the BIFT, which requires only one primary entry per neighbor. In the following we describe the necessary changes to use LFA-based BIER-FRR with the compact BIFT.

We propose to use a default BIFT that does not contain any backup entries and is used for forwarding in the failure-free case. In addition, we use failure-specific backup BIFTs. When a BFR detects that a specific neighbor is unreachable, it matches incoming packets on the backup BIFT that is associated with the unreachable NH. When the failure has been repaired or forwarding entries are updated, the BFR continues matching on the default BIFT.

2) Derivation of Backup BIFTs: We explain how the backup BIFT for a specific neighbor N is derived in two steps. First we fill the BIFT with entries and afterwards activate bits in specific F-BMs. We start with an empty backup BIFT. In the first step, for each neighbor that is not N, the corresponding primary entry from the default BIFT is added to the backup BIFT. In the second step, for each BFER B whose primary NH is N, LFAs are identified on the BIER topology. If an LFA is available, the bit that corresponds to B is activated in the F-BM of the BFR that is the LFA. If no LFA is available, B cannot be protected.

D. State Discussion

In a topology with n BFERs the normal BIFT contains n primary forwarding entries. LFA-based BIER-FRR requires n additional backup entries, which totals in $n + n$ forwarding entries. In contrast, the compact BIFT contains only one forwarding entry for each neighbor. Therefore, when the average node degree is k, the compact BIFT requires on average only k primary forwarding entries. On average each node has k backup BIFTs with on average $k - 1$ entries, which results in $k + k \cdot (k - 1)$ forwarding entries. Since the average node degree is significantly smaller than the number of destinations in a network, scalability of the compact BIFT is considerably better.

VIII. Comparison of LFA- and Tunnel-Based Protection for BIER

In this section we compare LFA-based and tunnel-based BIER-FRR. We point out similarities, and analyze protection capabilities and overhead with regard to header size and forwarding state. Afterwards, we discuss the impact of differences between Layer 3 topology and BIER topology.

A. Similarities

Both approaches implement FRR for BIER for resilient transport of IP multicast. Forwarding devices need to detect unreachable NHs, e.g. through a BFD. Both FRR mechanisms are based on pre-computed backup entries in addition to the primary forwarding entries. It is not necessary to change the structure of the BIFT. When the PLR cannot reach a primary NH, affected packets are rerouted according to the backup entries. Two modes of operation for link and node protection with different protection properties are available. For both, LFA- and tunnel-based BIER-FRR it is necessary to augment the forwarding procedure of BIER.

B. Protection Capabilities

We compare coverage properties and occurrence of loops.

1) Coverage: Tunnel-based BIER-FRR is able to protect traffic against arbitrary single component failures by design when the routing underlay provides full FRR coverage. As long as the destination is still reachable, an IP or segment routing tunnel is deployed to deliver the traffic to the unreachable NH or NNHs.

Protection of LFA-based BIER-FRR depends on the topology. The authors of [24] evaluate LP- and NP-LFAs on a comprehensive set of topologies. They find that LP-LFAs protect only 70% of destinations against single link failures and cause loops when nodes fail. NP-LFAs avoid loops when a node fails, but protect only 40% of destinations against single link failures. LP-rLFAs protect against any single link failure on unit link cost topologies. For any further guarantees TI-LFAs, or eLFAs have to be deployed. Both LFA extensions guarantee full protection for any single component failure in the network. However, augmenting LFA-based BIER-FRR with rLFAs, TI-LFAs, or eLFAs requires an additional header. TI-LFAs require an IP header stack, eLFAs require additional forwarding entries to implement backup paths.

2) Loops: Tunnel-based BIER-FRR cannot cause loops on the BIER layer because the packet is tunneled to the backup NH. When the packet is successfully delivered at the backup NH, BIER forwarding continues. If the tunnel is interrupted, the routing underlay is responsible for avoiding loops.

LFA-based BIER-FRR cannot guarantee to avoid loops because depending on the failure scenario and the mode of operation, all LFA variants can cause loops [24], [25]. With link protection, traffic may loop if at least one node or multiple links fail. With node protection, loops are prevented as long as not multiple components fails. In Section IV-B4 we review a loop detection mechanism for LFAs and all variants to prevent loops in any failure scenario. However, this mechanism significantly increases operational complexity and modifications to the packet header are necessary.

C. Overhead

We compare both protection approaches according to packet header size and required forwarding state.

1) Header Size: Tunnel-based BIER-FRR requires tunneling to protect traffic against failures. This adds an additional header to the packet. When the tunnel is interrupted and the routing underlay leverages a tunnel-based FRR protection mechanism for unicast, e.g. TI-. or eLFAs, an additional header is added to the packet. The basic form of the LFA-based BIER-FRR approach does not require tunneling. However, rLFAs, TI-LFAs, or eLFAs increase the protection capabilities of LFAs to an appropriate level but require at least one additional IP header. More header reduce the throughput and the Maximum Transmission Unit (MTU) has to be decreased at domain boundaries. The LFA-based approach requires a loop detection mechanism to prevent loops. Such a mechanism is available, however it increases packet header size even further.

2) Forwarding State: Both BIER-FRR approaches require the same amount of forwarding state. In a topology with n BFERs and an average node degree of k, the regular BIFT contains $n + n$ forwarding entries while the compact BIFT requires on average only $k + k \cdot (k - 1)$ entries. Since k is significantly smaller than n, deployment with the compact BIFT provides better scalability.

D. Influence of the BIER Topology

When some network nodes in a Layer 3 network do not support BIER, Layer 3 LFAs may disappear on the BIER layer. Thus, coverage of LFA-based BIER-FRR depends on the BIER topology. When regular LFAs have low coverage, LFA-based BIER-FRR needs to be complemented with rLFAs, TI-LFAs, or eLFAs. Backup paths may become longer in a sparse BIER topology because LFAs may be reachable only through a long Layer 3 tunnel. Tunnel-based BIER-FRR leverages tunnels through the routing underlay to the BIER NH or BIER NNHs for protection. Thus, tunnel-based BIER-FRR is not affected in a negative way by a BIER topology that is different from the Layer 3 topology.

IX. Conclusion

In this paper we compared LFA-based and tunnel-based BIER-FRR for resilient and scalable transport of IP multicast. Our discussion showed shortcomings of the LFA-based approach. Sometimes multiple packets are sent over one link, not all single link or single node failures can be protected, and in some scenarios backup traffic may loop. We propose extensions to overcome those shortcomings so that the capabilities of LFA-based and tunnel-based BIER-FRR mechanisms are equal. Differences remain in backup path length when the BIER topology is different from the Layer 3 topology, and in the need for additional headers.

References

[1] I. Wijnands, E. Rosen, A. Dolganow, T. Przygienda, and S. Aldrin, *RFC 8279: Multicast Using Bit Index Explicit Replication (BIER)*, https://datatracker.ietf.org/doc/rfc8279/, Nov. 2017.

[2] J. Papán, P. Segeč, M. Moravčík, M. Kontšek, L. Mikuš, and J. Uramová, "Overview of IP Fast Reroute solutions," in *ICETA*, 2019.

[3] I. Wijnands, G. J. Shepherd, C. J. Martin, and R. Asati, *Per-Prefix LFA FRR with Bit Indexed Explicit Replication*, https://patents.google.com/patent/US20180278470A1/en, Sep. 2018.

[4] G. Rétvári, J. Tapolcai, G. Enyedi, and A. Császár, "IP fast ReRoute: Loop Free Alternates revisited," in *IEEE INFOCOM*, 2011.

[5] D. Merling and M. Menth, *BIER Fast Reroute*, https://tools.ietf.org/html/draft-merling-bier-frr-00, Mar. 2019.

[6] A. Giorgetti, A. Sgambelluri, F. Paolucci, P. Castoldi, and F. Cugini, "First Demonstration of SDN-based Bit Index Explicit Replication (BIER) Multicasting," in *IEEE EuCNC*, 2017.

[7] K. C. Almeroth, "The Evolution of Multicast: From the MBone to Interdomain Multicast to Internet2 Deployment," *IEEE Network*, vol. 14, 2000.

[8] B. Zhang and H. T. Mouftah, "Forwarding State Scalability for Multicast Provisioning in IP Networks," *IEEE ComMag*, vol. 41, 2003.

[9] X. Li and M. J. Freedman, "Scaling IP Multicast on Datacenter Topologies," in *ACM CoNEXT*, 2013.

[10] S. Islam, N. Muslim, and J. W. Atwood, "A Survey on Multicasting in Software-Defined Networking," *IEEE Comst*, vol. 20, 2018.

[11] Z. AlSaeed, I. Ahmad, and I. Hussain, "Multicasting in Software Defined Networks: A Comprehensive Survey," *JNCA*, vol. 104, 2018.

[12] J. Rückert *et al.*, "Software-Defined Multicast for Over-the-Top and Overlay-based Live Streaming in ISP Networks," *JNSM*, vol. 23, 2015.

[13] J. Rückert, J. Blendin, and D. Hausheer, "Flexible, Efficient, and Scalable Software-Defined Over-the-Top Multicast for ISP Environments With DynSdm," *IEEE TNSM*, vol. 13, 2016.

[14] Y.-D. Lin, Y.-C. Lai, H.-Y. Teng, C.-C. Liao, and Y.-C. Kao, "Scalable Multicasting with Multiple Shared Trees in Software Defined Networking," *JNCA*, vol. 78, 2017.

[15] D. Kotani, K. Suzuki, and H. Shimonishi, "A Multicast Tree Management Method Supporting Fast Failure Recovery and Dynamic Group Membership Changes in OpenFlow Networks," *JIP*, vol. 24, 2016.

[16] T. Pfeiffenberger, J. L. Du, P. B. Arruda, and A. Anzaloni, "Reliable and Flexible Communications for Power Systems: Fault-tolerant Multicast with SDN/OpenFlow," in *IFIP NTMS*, 2015.

[17] A. Giorgetti, A. Sgambelluri, F. Paolucci, N. Sambo, P. Castoldi, and F. Cugini, "Bit Index Explicit Replication (BIER) Multicasting in Transport Networks," in *ONDM*, 2017.

[18] T. Eckert, G. Cauchie, W. Braun, and M. Menth, *Traffic Engineering for Bit Index Explicit Replication BIER-TE*, http://tools.ietf.org/html/draft-eckert-bier-te-arch, Nov. 2017.

[19] W. Braun, J. Hartmann, and M. Menth, "Demo: Scalable and Reliable Software-Defined Multicast with BIER and P4," in *IFIP/IEEE IM*, 2017.

[20] W. Braun, M. Albert, T. Eckert, and M. Menth, "Performance Comparison of Resilience Mechanisms for Stateless Multicast using BIER," in *IFIP/IEEE IM*, 2017.

[21] Z. Zhang and A. Baban, *Bit Index Explicit Replication (BIER) Forwarding for Network Device Components*, https://patents.google.com/patent/US20160191372, Dec. 2014.

[22] Q. Xiong, G. Mirsky, F. Hu, and C. Liu, *BIER BFD*, https://datatracker.ietf.org/doc/draft-hu-bier-bfd/, Oct. 2017.

[23] D. Katz and D. Ward, *Bidirectional Forwarding Detection (BFD)*, https://datatracker.ietf.org/doc/rfc5880/, Jul. 2004.

[24] W. Braun and M. Menth, "Loop-Free Alternates with Loop Detection for Fast Reroute in Software-Defined Carrier and Data Center Networks," *JNSM*, vol. 24, 2016.

[25] D. Merling, W. Braun, and M. Menth, "Efficient Data Plane Protection for SDN," in *IEEE NetSoft*, 2018.

[26] L. Csikor and G. Rétvári, "IP fast reroute with remote Loop-Free Alternates: The unit link cost case," in *ICUMT*, 2012.

[27] P. Francois, C. Filsfils, A. Bashandy, B. Decraene, and S. Litkowski, *Topology Independent Fast Reroute using Segment Routing*, https://tools.ietf.org/html/draft-francois-rtgwg-segment-routing-ti-lfa-00, Aug. 2015.

2.3 P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast

P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast

Daniel Merling[*], Steffen Lindner, Michael Menth

University of Tuebingen, Department of Computer Science, Chair of Communication Networks, Tuebingen, Germany

Abstract

Bit Indexed Explicit Replication (BIER) is a novel IP multicast (IPMC) forwarding paradigm proposed by the IETF. It offers a transport layer for other IPMC traffic, keeps core routers unaware of IPMC groups, and utilizes a routing underlay, e.g., an IP network, for its forwarding decisions. With BIER, core networks do not require dynamic signaling and support a large number of IPMC groups with large churn rates. The contribution of this work is threefold. First, we propose a novel fast reroute (FRR) mechanism for BIER (BIER-FRR) so that IPMC traffic can be rerouted as soon as the routing underlay is able to carry traffic again after a failure. In particular, BIER-FRR enables BIER to profit from FRR mechanisms in the routing underlay. Second, we describe a prototype for BIER and BIER-FRR within an IP network with IP fast reroute (IP-FRR). It is based on the programmable data plane technology P4. Third, we propose a controller hierarchy with local controllers for local tasks, in particular to enable IP-FRR and BIER-FRR. The prototype demonstrates that BIER-FRR reduces the protection time for BIER traffic to the protection time for unicast traffic in the routing underlay.

Keywords: Software-Defined Networking, P4, Bit Index Explicit Replication, Multicast, Resilience, Scalability

`https://doi.org/10.1016/j.jnca.2020.102764`

[*]Corresponding author

The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/2. The authors alone are responsible for the content of the paper.

Email addresses: `daniel.merling@uni-tuebingen.de` (Daniel Merling), `steffen.lindner@uni-tuebingen.de` (Steffen Lindner), `menth@uni-tuebingen.de` (Michael Menth)

1. Introduction

IP multicast (IPMC) is leveraged for many services like IPTV, multicast VPN, or the distribution of financial or broadcast data. It efficiently forwards one-to-many traffic on tree-like structures to all desired destinations by sending at most one packet copy per link in the distribution tree. IPMC is organized into sets of receivers, so-called IPMC groups. Hosts subscribe to IPMC groups to receive the traffic which is addressed to that group. Traditional IPMC methods require per-IPMC-group state within core routers to forward the packets to the right next-hops (NHs). This raises three scalability issues. First, the number of IPMC groups may be large which require lots of space in forwarding tables. Second, core routers are involved in the establishment, removal, and in the change of an IPMC group. This requires significant signaling in the core network every time subscribers change because many nodes possibly need to update their forwarding information base, which imposes heavy load on core when churn rates are large. Third, when the topology changes or in case of a failure, the forwarding of any IPMC group possibly requires fast update, which is demanding in a critical network situation. IPMC features are available in most off-the-shelf hardware, but the features are turned off by administrators due to complexity and limited scalability.

The Internet Engineering Task Force (IETF) developed Bit Index Explicit Replication (BIER) [1, 2] as a solution to those problems. BIER features a domain concept. Only ingress and egress routers of a BIER domain participate in signalling. They encapsulate IPMC packets with a BIER header containing a bit string that indicates the receivers of the IPMC group within the BIER domain. Based on that bit string the packets are forwarded through the BIER domain without requiring per-IPMC-group state in core routers.

BIER leverages the so-called bit indexed forwarding table (BIFT) for forwarding decisions. Its entries are derived from paths induced by the interior gateway protocol (IGP) which is used for unicast routing. In the following we refer to that routing protocol with the term 'routing underlay'. Therefore, BIER traffic follows the same paths as the unicast traffic carried by the routing underlay. So far, BIER lacks any protection capabilities. In case of a link or node failure, the BIFT entries need to be changed so that BIER traffic is carried around failed elements towards receivers. However, the BIFTs can be updated only after the routing underlay has updated its forwarding information base and based on the new paths, BIER forwarding entries are recomputed. This takes a significant amount of time. In the meantime, packets are dropped. When a multicast packet is dropped, all downstream subscribers cannot receive the packet. Regular IP forwarding is affected as well by failures, but for unicast traffic, fast reroute (FRR) [3] mechanisms have been proposed to reroute affected packets on backup paths until the primary forwarding entries are updated. IP-FRR leverages pre-computed backup actions for fast recovery in case of a failure without the need for signaling. However, IP-FRR is not a suitable protection method for multicast traffic because it does not consider the tree-like forwarding structures along which IPMC packets are distributed.

In this work, we introduce BIER-FRR. It has two different operation modes to protect either only against link failures or also against node failures. We recently proposed this mechanism in the BIER working group of the IETF [4]. BIER has been suggested as a novel transport mechanism for IPMC. However, it cannot be configured yet on standard hardware. New, programmable data plane technologies allow the definition of new packet headers and forwarding behavior, and offer themselves for the implementation of prototypes. In [5], we presented an early version of a P4-based prototype for BIER. It was based on the P_{14} specification of P4 [6] and required a few workarounds because at that time some P4 features were not available on our target, the software switch BMv2. Moreover, there was no protection method available for BIER. We now provide the description of a completely reworked prototype on the base of the P_{16} specification of P4 [7]. The new prototype implements IP forwarding, a simple form of IP-FRR, BIER forwarding, and BIER-FRR. It comprises a controller hierarchy with local controllers that enables FRR techniques with P4. We argue that local controllers are needed for protection and helpful for local tasks in general. The evaluation of the prototype shows that BIER traffic is not longer affected by network failures than unicast traffic when BIER-FRR is enabled. Thus, the contribution of this paper is threefold: (1) a concept for BIER-FRR, (2) an implementation of BIER and BIER-FRR with P4, and (3) a controller hierarchy with local controllers to support FRR techniques in P4. Finally, the P4-based prototype demonstrates the usefulness of BIER-FRR. The source code of our prototype with the fully working data and control plane is publicly available on GitHub.

The remainder of this paper is structured as follows. Section 2 reviews basics of multicast. Section 3 contains fundamentals about IP-FRR, explains why it is insufficient to protect multicast traffic, and examines related work. Section 4 discusses related work for both legacy- and SDN-based multicast. Section 5 gives a primer on BIER. Section 6 explains the resilience problem of BIER and introduces BIER-FRR. In Section 7 we summarize necessary basics of P4 needed for the understanding of the BIER prototype. Section 8 describes the P4-based prototype implementation of IP, IP-FRR, BIER, and BIER-FRR. The prototype is used to demonstrate the usefulness of BIER-FRR in Section 9. Finally, Section 10 summarizes this paper and discusses further research issues.

2. Technological Background for IP Multicast

This section gives a primer on IP multicast (IPMC) for readers that are not familiar with IPMC. IPMC supports one or more sources to efficiently communicate with a set of receivers. The set of receivers is called an IPMC group and is identified by an IP address from the Class D address space (224.0.0.0 – 239.255.255.255). Devices join or leave an IPMC group leveraging the Internet Group Management Protocol (IGMP) [8].

IPMC packets are delivered over group-specific distribution trees which are computed and maintained by IPMC-capable routers. In the simplest form, source-specific multicast trees based on the shortest path principle are computed and installed in the routers. The notation (S, G) identifies such a shortest path tree for the source S and the group G.

IPMC also supports the use of shared trees that can be used by multiple senders to send traffic to a multicast group. The shared tree has a single root node called rendezvous point (RP). The sources send the multicast traffic to the RP which then distributes the traffic over a shared tree. In the literature, shared trees are denoted by $(*, G)$.

Protocol-independent multicast (PIM) leverages unicast routing information to perform multicast forwarding. PIM cooperates with various unicast routing protocols such as OSPF or BGP and supports both source-specific and shared multicast distribution trees.

Pragmatic General Multicast (PGM) [9] reduces packet loss for multicast traffic. It enables receivers to detect lost or out-of-order packets and supports retransmission requests similar to TCP.

3. IP Fast Reroute

In this section we give a primer on IP fast reroute (IP-FRR). First, we explain fundamentals of IP-FRR and describe Loop-Free Alternates. Then, we discuss related work.

3.1. Fundamentals of IP-FRR

When a link or a node fails, devices may not be able to forward packets to their NHs. As soon as a failure is detected in an IP network, the changed topology is signaled through the network, new working paths are calculated, and the forwarding tables of all devices are consistently updated. This process is called reconvergence and may take up to several seconds. In the meantime, packets are dropped in case of wrong or missing forwarding entries. IP-FRR [3] protects IP unicast traffic against the failure of single links and nodes while reconvergence is ongoing. It is based on pre-computed backup actions to quickly reroute affected packets. Figure 1 shows an example for Loop-Free Alternates (LFAs) [10] which is the most popular IP-FRR mechanism. When a node's

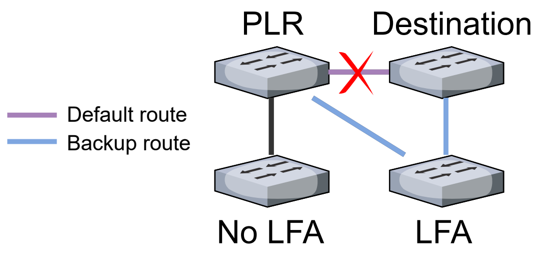


Figure 1: A PLR reroutes a packet to a backup path when the NH on the primary path is unreachable.

(primary) NH becomes unreachable, the node detects that failure after some time and reroutes the traffic locally over a backup path. Therefore, the node is also called point of local repair (PLR). To reroute packets in a timely manner, nodes store a backup NH in addition to the primary NH for each destination. When the PLR detects that the primary NH is unrechable, e.g., by loss-of-light detection, loss-of-carrier detection, a bidirectional forwarding detection[1] (BFD) [11], or any other suitable mechanism, it forwards the packet to its backup NH instead. That backup NH is called Loop-Free Alternate (LFA) and it has to be chosen such that rerouted traffic does not loop. However, some destinations remain unprotected because there is not always an alternative hop that has a shortest path towards the destination which avoids the failed element. The set of protected destinations is also called coverage. The limited coverage of LFAs has been evaluated in various studies [12, 13].

3.2. Related Work

The two surveys [14, 15] give an overview of several IP-FRR mechanisms. We discuss only some of the papers. Equal-cost multi-path (ECMP) can be used as a very basic FRR mechanism. When a PLR has at least two paths with equal cost towards a destination, it quickly deviates traffic to the other path when the primary NH is unreachable. However, this works only if two equal-cost paths are available under normal conditions, which is mostly not the case. Not-via addresses [16, 13] tunnel packets to the downstream next-next-hop (NNH) if the NH is unreachable. To that end, the NNH is assigned a unique address and an explicit backup path is constructed which does not include the failed component. Loop-Free Alternates (LFAs) [10] forward packets to alternative NHs if the primary NH is unreachable. Those alternative NHs have to be chosen in a way that they have a working shortest path to the destination that avoids rerouting loops. As such alternative neighbors do not exist for all PLRs and destinations, the LFA mechanism cannot protect against all single link and node failure. Remote LFAs [17] (rLFAs) extend the protection capabilities of LFAs by sending affected packets through shortest path tunnels to nodes that still reach the destination on a working shortest path. rLFAs protect against any single link failure in unit link cost networks. However, they achieve only partial coverage in case of node failures or non-unit link costs. An analysis can be found in [12].

4. Related Work

We review work in the context of SDN-based multicast. Most traditional multicast approaches were implemented with OpenFlow. Some works considered protection mechanisms. A few studies improve the efficiency of multicast forwarding with SDN. Only a single work implements BIER without protection using OpenFlow, but the implementation itself requires dynamic forwarding state, which runs contrary to the intention of BIER.

[1]When a BFD is established between two nodes, they periodically exchanges keep-alive notifications.

4.1. Multicast Implementations with OpenFlow

The surveys [18, 19] provide an extensive overview of multicast implementations for SDN. They discuss the history of traditional multicast and present multiple aspects for SDN-based multicast, e.g., multicast tree planning and management, multicast routing and reliability, etc. In the following we briefly discuss some exemplary works that implement multicast for SDN. More details can be found in the surveys or the original papers.

Most related works with regard to SDN-based multicast implement explicit flow-specific multicast trees. Most authors propose to compute traffic-engineered multicast trees in the controller using advanced algorithms and leverage SDN as tool to implement the multicast path layout. The following works provide implementations in OpenFlow to show the feasibility of their approaches. Dynamic Software-Defined Multicast (DynSDM) [20, 21] leverages multiple trees to load-balance multicast traffic and efficiently handle group subscription changes. Modified Steiner tree problems are considered in [22, 23] to minimize the total cost of edges and the number of branch nodes, or to additionally minimize the source-destination delay [24]. In [25], the authors compute and install traffic-engineered shared multicast trees using OpenFlow. In [26] and [27], traffic-engineered Steiner trees are computed which minimize the number of edges of the tree and provide optimized locations for multicast sources in the network. The Avalanche Routing Algorithm (AvRA) [28] considers topological properties of data center networks to optimize utilization of network links. Dual-Structure Multicast (DuSM) [29] improves scalability and link utilization for SDN-based data center networks by deploying different forwarding approaches for high-bandwidth and low-bandwidth flows. In [30], Steiner trees are leveraged to compute a multicast path layout including certain recovery nodes which are used for reliable multicast transmission such as PGM. In [31], the authors evaluate different node-redundant multicast tree algorithms in an SDN context. They evaluate the number of forwarding rules required for each mechanism and study the effects of node failures. The authors of [32] reduce the number of forwarding entries in OpenFlow switches for multicast. They propose to use address translation from the multicast address to the receiver's unicast address on the last branching node of the multicast tree. This allows to omit multicast-specific forwarding entries in leaf switches.

4.2. Multicast Protection with OpenFlow

Kotani et al. [33] suggest to utilize primary and backup multicast trees for SDN networks. Multicast packets carry an ID to identify the distribution tree over which they are forwarded. In case of a failure, the controller chooses an appropriate backup multicast tree and reconfigures senders accordingly. This mechanism differs in two ways from BIER-FRR. First, the controller has to be notified, which is not suitable for fast recovery. Second, it requires significant signalling effort in response to a failure.

The authors of [34] propose a FRR method for multicast in OpenFlow networks. Multicast traffic is forwarded along a default distribution tree. If a downstream neighbor is unreachable, traffic is switched to a backup distribution tree that contains all downstream nodes of the unreachable default subtree. The backup distribution tree must not contain the unreachable neighbor as forwarding node. VLAN tags are used to indicate the trees over which multicast packets should be sent. This mechanism differs from BIER-FRR in a way that it requires a significant amount of additional dynamic forwarding state to configure the backup trees.

4.3. Improved Multicast Forwarding for SDN Switches

Some contributions improve the efficiency of devices to enable hardware-based multicast forwarding. The work in [35] organizes forwarding entries of a switch based on prime numbers and the Chinese remainder theorem. It reduces the internal forwarding state and allows for more efficient hardware implementations. Reed et al. provide stateless multicast switching in SDN-based systems using Bloom filters in [36] and implement their approach for TCAM-based switches. The authors compare their approach with existing layer-2 forwarding and show that their method leads to significantly lower TCAM utilization.

4.4. SDN Support for BIER

The authors of [37, 38] implement two SDN-based multicast approaches using (1) explicit multicast tree forwarding and (2) BIER forwarding in OpenFlow. They realize explicit multicast trees with OpenFlow group tables. To support BIER, they leverage MPLS headers to encode the BIER bit string, which limits the implementation to bit strings with a length of 20 bits, and therefore a maximum of 20 receivers. Rules with exact matches for bit strings are installed in the OpenFlow forwarding tables. When a packet with a BIER header arrives at a switch and a rule for its bit string is available, the packet can be immediately transmitted over the indicated interfaces. Otherwise, a local BIER agent running on the switch and maintaining the BIFT is queried. The local BIER agent installs a new flow entry for the specific bit string in the OpenFlow forwarding table. Thus, this approach requires bit string-specific state instead of IPMC group specific state. Furthermore, it is not likely to work well with quickly changing multicast groups as most subscription changes require configuration changes in the forwarding table of the switch. BIER with support for traffic engineering (TE) has been proposed in [39]. It uses the same header format but features different forwarding semantics and is not compatible with normal BIER. In [40] we have proposed and evaluated several FRR algorithms for BIER-TE and implemented them in a P4-based prototype [5].

5. Bit Index Explicit Replication (BIER)

First, we give an overview of BIER [2]. Afterwards, we present the Bit Index Forwarding Table (BIFT), which is the forwarding table for BIER. Then, we describe the BIER forwarding procedure.

5.1. *Overview*

We introduce essential nomenclature for BIER, the layered BIER architecture, the BIER header, and the BIER forwarding principle.

5.1.1. *BIER Domain*

BIER leverages a domain concept to transport IPMC traffic in a scalable manner, which is illustrated in Figure 2. Bit-Forwarding Routers (BFRs) forward BIER multicast traffic within the BIER domain. Inbound multicast traffic enters the domain through Bit-Forwarding Ingress Routers (BFIRs) and leaves it through Bit-Forwarding Egress Routers (BFERs). Border routers usually implement both BFIR and BFER functionality. When a BFIR receives an inbound IPMC packet, it pushes a BIER header onto the IPMC packet which indicates all BFERs that should receive a packet copy. BFRs utilize the information in the BIER header to forward BIER packets to all desired destinations along the paths induced by the interior gateway protocol (IGP). Thereby, packets are replicated if needed. Finally, the BFERs remove the BIER header before forwarding IPMC packets outside the domain.

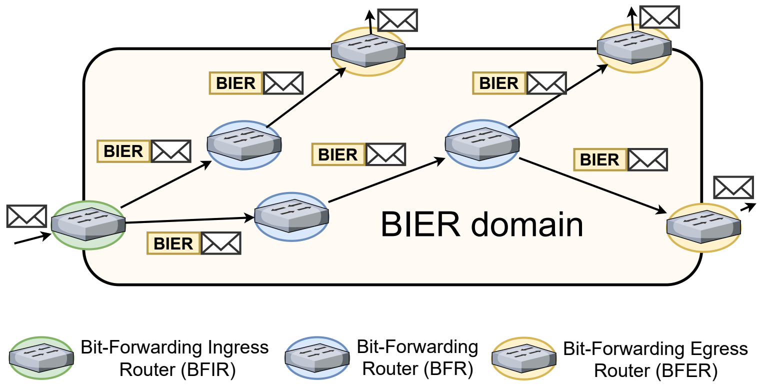

BIER
BIER
BIER
BIER
BIER
BIER
BIER domain
Bit-Forwarding Ingress Router (BFIR)
Bit-Forwarding Router (BFR)
Bit-Forwarding Egress Router (BFER)

Figure 2: IPMC traffic enters a BIER domain through BFIRs which equip it with a BIER header. BFRs forwarded the traffic based on the BIER header within the domain on paths induced by the IGP. BFERs remove the BIER header when the traffic leaves the domain.

5.1.2. *A Layered BIER Architecture*

The BIER architecture can be subdivided into three layers: the IPMC layer, the BIER layer, and the routing underlay which is the forwarding mechanism for unicast traffic. In IP networks, the latter corresponds to the interior gateway protocol (IGP). Figure 3 shows the relation among the layers.

The IPMC layer requests multicast delivery for IPMC packets to a set of receivers in a BIER domain that depend on IPMC subscriptions. That is, when an inbound IMPC packet arrives at a BFIR, the BFIR equips the IPMC packet with an appropriate BIER header indicating all desired destinations. The BIER layer forwards these packets through the BIER domain to all receivers indicated

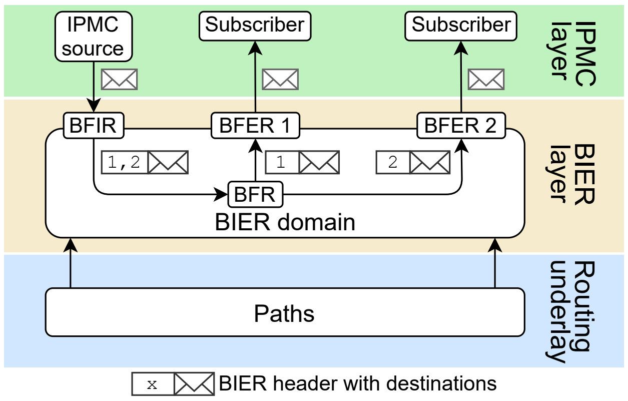


Figure 3: Layered BIER architecture with IPMC layer, BIER layer, and the routing underlay.

in the BIER header, thereby implementing a stateless point-to-multipoint tunnel für IPMC. The BIER layer leverages the forwarding information of the routing underlay to populate the forwarding tables of the BFRs. As a result, BIER traffic to a specific BFER follows the same path as unicast traffic towards that BFER. If two BFR are connected on Layer 2, the BIER traffic is directly forwarded; otherwise, the BFR neighbor is reachable only over the routing underlay so that the BIER traffic is encapsulated and forwarded over the routing underlay. When a BIER packet reaches a BFER that should receive a packet copy, the BFER removes the BIER header and passed the IPMP packet to the IPMC layer for further forwarding.

5.1.3. BIER Header and Forwarding Principle

The BIER header contains a bit string to identify BFERs. For brevity, we talk in the following about the BitString of a packet to refer to the bit string in the BIER header of that packet. The BitString is of a specific lenght. Each bit in the BitString corresponds to one specific BFER. The bits are assigned to BFERs starting with the least significant bit. BIER devices must support a BIER header of 256 bits. As this may not suffice to assign bits to all BFERs in large networks, the standard [2] defines subdomains to cope with that problem. This is a technical detail that we do not consider any further and our proposed solution can be easily adapted to subdomains.

When a BFIR receives an IPMC packet, it pushes a BIER header to the IPMC packet, determines the set of BFERs that must receive the traffic of the respective IMPC group, and activates in the BitString the bits corresponding to these BFERs. Packets are forwarded based on the information in their BIER header. A BFR sends a packet to any of its interfaces if at least one BFER indicated in the BIER header is reached in the routing underlay over this specific interface. To avoid duplicates, only those bits are kept in the BitString whose BFERs can be reached over the specific interface.

Figure 4 illustrates the BIER forwarding principle. It shows a small BIER domain with four nodes, each of them being BFR, BFIR, and BFER. Hosts are attached to all BIER nodes and participate in a single multicast group. Host 1 sends an IPMC packet to all other hosts. The figure visualizes how the BitString changes along the forwarding tree whose paths are inherited from the routing underlay.

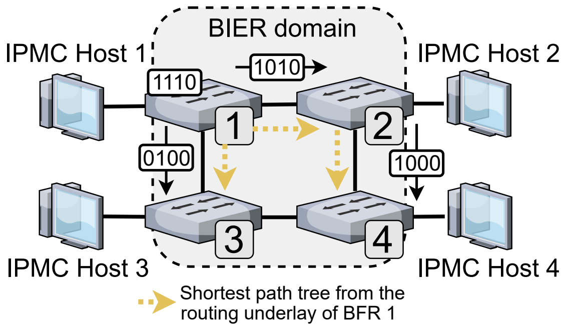


Figure 4: An IPMC packet is forwarded from Host 1 to all other hosts via a BIER domain. Within the domain, BIER packets are forwarded based on the BitString.

The information of the BIER forwarding tables depends only on the routing underlay. In Section 5.2 we explain the structure of the table and how its entries are calculated. In contrast to traditional IPMC forwarding, BIER forwarding does not require knowledge about IPMC groups. This has several advantages. BFRs do not neet to keep state per IPMC group. This frees core nodes of a BIER domain from signalling and state processing per IPMC group when subscriptions or routes change, e.g., in case of failures. This makes BIER forwarding in core nodes more robust and scalable than traditional IPMC forwarding. BFIRs still participate in IPMC signaling to keep track of group changes in order to adapt the BIER header for each IPMC group. BFERs forward outbound IPMC traffic in a traditional way.

5.2. Bit Index Forwarding Table

In this section we describe the Bit Index Forwarding Table (BIFT) which is the forwarding table of BFRs. We explain its structure and the computation of its entries.

First, we define BFR neighbors (BFR-NBRs) before we introduce the structure of the BIFT. The BFR-NBRs of a BFR A are those BFRs, that are adjacent to A according to the paths of the routing underlay.

Each BFR maintains a Bit Index Forwarding Table (BIFT) to determine the NH, i.e., BFR-NBR, when forwarding a packet. Table 1 shows the structure of the BIFT. For each BFER, the BIFT contains one entry which consists of a forwarding bitmask (F-BM) and the BFR-NBR to which the packet should be sent. The F-BM is used in the forwarding process to clear bits in a packet's BitString before transmission. The BFR-NBR for a BFER is derived as the

BFER	F-BM	BFR-NBR

Table 1: Header of the BIFT.

BFR-NBR on the path from the considered BFR to the BFER in the routing underlay. The F-BM for a BFER is composed as a bit string where all bits are activated that belong to BFERs with the same BFR-NBR. As a result, all BIFT entries with the same BFR-NBRs also have the same F-BM.

Table 2 illustrates the BIFT of BFR 1 in the example given in Figure 4.

BFER	F-BM	BFR-NBR
1	-	-
2	1010	2
3	0100	3
4	1010	2

Table 2: BIFT of BFR 1.

5.3. BIER Forwarding

In this paragraph we describe BIER forwarding. First, we explain the procedure how BIER processes packets. Then, we show a forwarding example. Finally, we illustrate the BIER header stack.

5.3.1. BIER Forwarding Procedure

BFRs process BIER packets in a loop. When a BFR receives a BIER packet, it determines the position of the least-significant activated bit in the BitString. The position of that bit corresponds to a BFER which is processed in this specific iteration of the loop. The BFR looks up that BFER in the BIFT, which results in a BFR-NBR and a F-BM. Then, a copy of the BIER packet is created for transmission to that BFR-NBR. Before transmission, all bits are cleared in the BitString of the packet copy that are not reachable through the same BFR-NBR. This is achieved by an AND-operation of the BitString and the F-BM. We denote this action as "applying the F-BM to the BitString". Then, the packet copy is forwarded to the indicated BFR-NBR. All BFERs in the IPMC subtree of that BFR-NBR eventually receive a copy of this sent packet if their bit is activated in the BitString of the packet copy. Thus, all BFERs of this IMPC subtree can be considered as processed. Therefore, their bits are removed from the BitString of the remaining BIER packet. To that end, the BFR applies the complement of the F-BM to the BitString of the remaining BIER packet. This ensures that packets are delivered only once to intended receivers. If the BitString in the remaining BIER packet still contains activated bits, the loop restarts; otherwise the processing loop stops.

When the BFER that is currently processed corresponds to the BFR itself, the F-BM and BFR-NBR of its BIFT entry are empty. Then, a copy of the

BIER packet is created, the BIER header is removed, and the packet is passed to the IPMC layer within the BFR. Afterwards, the processed bit is cleared in the BitString of the remaining BIER packet, and the loop restarts if the BitString contains activated bits; otherwise the loop stops.

5.3.2. BIER Forwarding Example

We assume that BFIR 1 in Figure 4 receives an IPMC packet from IPMC Host 1 that should be sent to the IPMC Hosts 2, 3, and 4. Therefore, it adds a BIER header with the BitString `1110` to the IPMC packet and processes it. The least-significant activated bit corresponds to BFR 2. BFR 1 looks up the activated bit in its BIFT which is shown in Table 2. Then, it creates a packet copy and applies the F-BM to the BitString of that copy. This sets the BitString to `1010`. Then, the packet copy is forwarded to BFR 2. Aftwards, BFR 1 clears the activated bits of the F-BM from the BitString of the remaining original BIER packet. This leaves a packet with the BitString `0100`. BFR 1 processes the next activated bit, i.e. the bit for BFER 3. A packet copy is created, and the F-BM is applied which leaves the BitString `0100` in the packet copy. Then it is forwarded to BFR 3.

BFR 2 process the packet in the same way. As a result, it forwards one packet copy with the BitString `1000` to BFR 4. Additionally, it sends an IPMC packet without BIER header to its connected host. BFR 3 and 4 do the same when they receive their respective BIER packet.

5.3.3. BIER Header Stack

Without loss of generality, we assume in the following that the routing underlay is IP. Furthermore, we neglect the role of Layer 2 to facilitate readability.

Each BIER device is also an IP device. However, not every IP device is a BIER device. In Figure 5, the “Pure IP-node” is an IP node without BIER functionality. It belongs to the IP topology but not to the BIER topology. This influences the header stack of forwarded BIER packets. BFR 1, 2 and 3 are

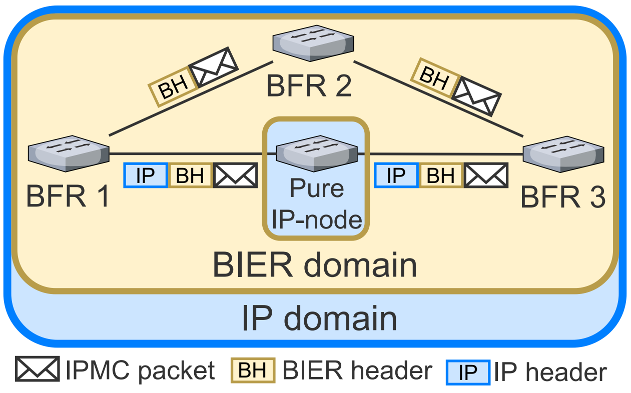


Figure 5: BIER traffic forwarded over pure IP nodes requires additional IP encapsulation.

both IP and BIER devices. The three BFRs are BFR-NBRs to each other. BFR 1 and 2 are neighbors to each other in both the IP and BIER topology because they are directly connected on Layer 2. Therefore, they exchange BIER packets on Layer 2 without an additional header. Since the pure IP node is not part of the BIER topology, BFR 1 and BFR 3 are BFR-NBRs although they are not neighbors in the IP topology. To exchange packets, BFR 1 and BFR 3 encapsulate BIER packets with an IP header and forward them via the pure IP node. When BFR 1 or 3 receive the packet, they remove the IP header and process the BIER header.

6. BIER Fast Reroute

The necessity for resilience mechanisms in BIER networks has been discussed in [41] without proposing any mechanism. In this section we introduce BIER fast reroute (BIER-FRR) to protect BIER traffic against link and node failures by taking advantage of reconvergence and FRR mechanisms of the routing underlay. We explain why regular BIER cannot protect BIER traffic sufficiently against failures, and present BIER-FRR for link and node protection, respectively. Finally, we discuss the protection properties of BIER-FRR.

6.1. Link Protection

In this paragraph we introduce BIER-FRR with link protection. First, we explain why relying on the available features of BIER and a resilient routing underlay is not sufficient for protection against link failures. Afterwards, we describe BIER-FRR with link protection and show a forwarding example.

6.1.1. Resilience Problems of BIER for Link Failures

BFR-NBRs may be directly connected over Layer 2 or they may be reachable only over Layer 3 so that IP encapsulation is needed for them to exchange BIER traffic (see Section 5.3.3). This has impact on the effect of link failures.

If neighboring BFRs are reachable only over Layer 3, they exchange BIER traffic IP-encapsulated towards each other. If a link on the path towards the BFR-NBR fails, the BFR-NBR is not reachable until the routing underlay has restored reachability. This may be due to IP-FRR, which is fast, or IP routing reconvergence, which is slow. In any case, the reachability on the BIER layer is also restored and no further action needs to be taken. Possibly, the path in the routing underlay changed, which may affect the neighboring relationships among BFRs, so that BIFTs need to be recomputed. This, however, is not time-critical.

If BFR-NBRs are directly connected over Layer 2, they exchange packets without an additional IP header. If the link between them is broken, protection mechanisms on Layer 3, in particular IP-FRR, cannot help because the BIER packet is not equipped with an IP header. Therefore, affected BIER traffic cannot be forwarded until a new BFR-NBR is provided in the BIFT for affected BFERs. Thus, the BIFT needs to be updated. This process takes time to recompute the entries based on the new paths from the routing underlay and starts only after reconvergence of the routing underlay has completed. This is significantly later than FRR mechanisms on the routing underlay restore connectivity for unicast traffic.

BIER-FRR with link protection effects that a BFR affected by a link failure can forward BIER traffic again as soon as its connectivity problem is detected on the BIER layer and the routing underlay is able to forward unicast traffic again.

6.1.2. BIER-FRR with Link Protection

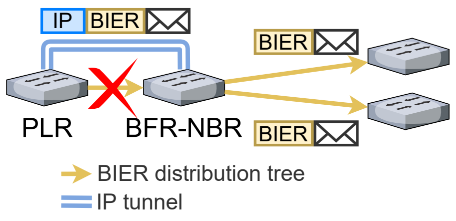


Figure 6: BIER-FRR with link protection is needed for BFR-NBRs which are directly connected on Layer 2: they IP-encapsulate BIER traffic towards a BFR-NBR after it is detected unreachable.

The concept of BIER-FRR with link protection is illustrated in Figure 6. BFRs must be able to detect link failures. This may happen, e.g., through loss of light detection or through bidirectional forwarding detection (BFD) with BFR-NBRs [42]. If an unreachable BFR-NBR is detected, a BFR IP-encapsulates BIER traffic towards that BFR-NBR. As a result, the BIER traffic will reach the affected BFR-NBR again as soon as reachability on the routing underlay is restored. This can be very fast if the routing underlay leverages FRR. When the traffic arrives at the BFR-NBR, the additional IP header is removed and packets are processed as normal BIER traffic.

6.1.3. Example for BIER-FRR with Link Protection

Figure 7 shows the BIER topology from the earlier forwarding example in Figure 4 with a link failure. For convenience, the BIFT of BFR 1 is displayed again in Table 3.

When BFR 1 sends a BIER packet to all other BFERs, the BitString is `1110`. First a packet copy is successfully deliverd to BFER 2 and BFER 4 so that the BitString of the remaining packet is `0100`, i.e., next a packet must be forwarded to BFER 3. However, BFR-NBR 3 is unreachable for BFR 1 due to the link failure. Therefore, BFR 1 IP-encapsulates the BIER packets towards BFR 3. As soon as the routing underlay restores connectivity, the IP-encapsulated BIER packets is detoured via BFR 2 and BFR 4 towards BFR 3. Thus, BIER-FRR with link protection may send a second packet copy over a link.

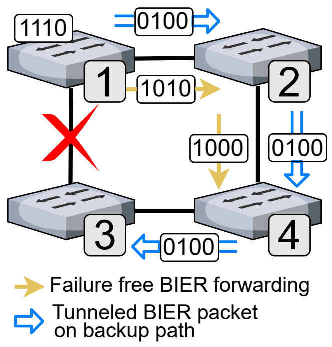

1110
0100
1
2
1010
1000
0100
3
4
0100
Failure free BIER forwarding
Tunneled BIER packet on backup path

Figure 7: Packet paths and example topology for BIER-FRR with link protection.

BFER	F-BM	BFR-NBR
1	-	-
2	1010	2
3	0100	3
4	1010	2

Table 3: BIFT of BFR 1.

6.2. Node Protection

We introduce BIER-FRR with node protection. First, we discuss why regular BIER cannot protect BIER traffic sufficiently fast against node failures. Afterwards, we present the concept of BIER-FRR with node protection, extend the BIFT with backup entries, show a forwarding example, and explain how backup entries are computed.

6.2.1. Resilience Problems of BIER for Node Failures

If a BFR fails, all downstream BFRs are unreachable. This problem cannot be quickly repaired by the routing underlay because traffic directed to the failed node cannot be delivered. Thus, alternate BFR-NBRs are needed. These are provided when the routing underlay has reconverged and the BIFT entries are recomputed. This, however, may take long time.

BIER-FRR with node protection shortens this time so that affected BIER traffic can be delivered in the presence of node failures as soon as connectivity for unicast traffic is restored in the routing underlay.

6.2.2. BIER-FRR with Node Protection

We propose BIER-FRR with node protection to deliver BIER traffic even if the BFR-NBR fails. The concept is shown in Figure 8. When a PLR cannot reach a BFR-NBR, it tunnels copies of the BIER packet to all BFR next-next-hops (BFR-NNH) in the distribution tree that should receive or forward a copy. Thus, for each such BFR-NNH, an individual packet copy is created. The packet is then tunneled to the BFR-NNH with an additional header of the routing underlay; these packets are delivered as soon as the routing underlay restores connectivity. When the BFR-NNH receives such a packet, it removes the tunnel header and processes the resulting BIER packet.

If a BFR-NBR is unreachable, the link towards the BFR-NBR or the BFR-NBR itself may have failed. Therefore, the BFR-NBR should also receive a packet copy encapsulated by the routing underlay.

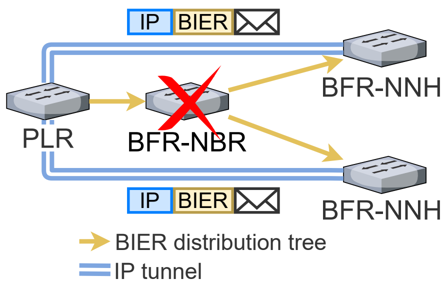


Figure 8: Concept of BIER-FRR with node protection.

BFER	F-BM	BFR-NBR
1	primary F-BM backup F-BM	primary NH backup NH
...	...	...

Table 4: Structure of a BIFT with backup entries.

When a packet copy is sent to multiple BFR-NNHs instead of the BFR-NBR, the the BitString of the forwarded packet copies must be modified appropriately to avoid duplicate packets at BFERs. These modifications can be obtained with backup F-BMs, which is explained in more detail in Section 6.2.5.

6.2.3. BIFT with Backup Entries

To support BIER-FRR with node protection, the BIFT must be extended with backup entries. The structure of a BIFT with backup entries is shown in Table 4.

The normal BIFT entries are called primary entries. The backup entries have the same structure as the primary entries. When a BFR-NBR is reachable, the primary entries are used for forwarding. If a BFR-NBR becomes unreachable, the corresponding backup entry is used for forwarding in the same way as a primary entry with only one difference. The packet is not forwarded natively but instead it is always tunneled to the backup NH through the routing underlay.

6.2.4. Example for BIER-FRR with Node Protection

Figure 9 shows an example topology and Figure 10 illustrates the distribution tree for BFR 1 and BFR 2 based on the paths from the routing underlay. Table 5 shows the BIFT of BFR 1 with primary and backup entries.

We illustrate the forwarding with BIER-FRR when BFR 2 fails. We assume that BFR 1 needs to send a BIER packet to BFR 6, i.e. the packet contains the BitString `100000`. As BFR 2 is unreachable, the primary NH of BFR 1 to BFR 6, which is BFR 2, cannot be used. Therefore, the backup entry is utilized. That means, the backup F-BM `101000` (see Table 5) is applied to the copy of the BIER packet and then it is tunneled through the routing underlay to the backup NH BFR 4. BFR 1 applies the complement of the backup F-BM

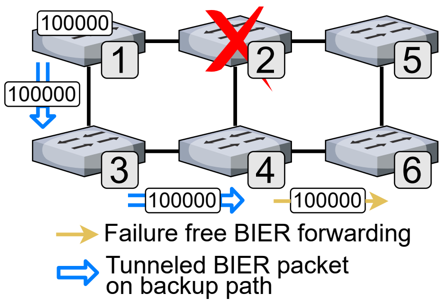

100000
1
2
5
100000
3
4
6
100000
100000
Failure free BIER forwarding
Tunneled BIER packet on backup path

Figure 9: A packet is sent from BFR 1 to BFR 6 over a backup path using node protection.

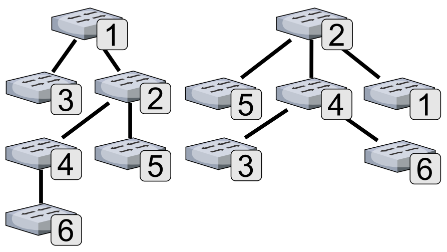


Figure 10: Shortest-path tree of BFR 1 and BFR 2.

BFER	F-BM	BFR-NBR
1	000001 -	- -
2	111010 000010	2 2
3	000100 000100	3 3
4	111010 101000	2 4
5	111010 010000	2 5
6	111010 101000	2 4

Table 5: BIFT of BFR 1 with primary and backup entries.

`010111` to the BitString of the original BIER packet which is then `000000`. As the BitString of the remaining BIER packet has no activated bits anymore, the forwarding process terminates at BFR 1.

The routing underlay delivers the packet copy from BFR 1 to BFR 4 as soon as connectivity is restored. BFR 4 removes the tunnel header and forwards the BIER packet to BFR 6.

If the BitString of the packet was `100100`, i.e., BFER 3 should have received a copy of the packet, too, a regular BIER packet would have been forwarded directly to BFR 3 before BIER-FRR tunnels another copy of the BIER packet to BFR 4. Thus, BIER-FRR with node protection may increase the traffic on a link to ensure that all relevant NNHs receive a packet copy.

6.2.5. Computation of Backup Entries

We compute backup NHs and backup F-BMs for BFERs at a specific BFR which we call PLR in this context. To that end, we distinguish two cases: the

BFER is not a BFR-NBR (1) or it is a BFR-NBR (2).

In the first case, the BFER is reached from the PLR through the routing underlay via a considered NH and next-next-hop (NNH). The considered NNH becomes the backup NH for the BFER. The corresponding backup F-BM requires activated bits for a set of BFERs. This set comprises all BFERs whose paths in the routing underlay from the PLR also traverses the considered NH and NNH. This F-BM can be computed by bitwise AND'ing the PLR's F-BM for the considered BFER and the considered NH's F-BM.

In the second case, the considered BFER is a BFR-NBR. Then, the NH is also taken as backup NH. This ensures that the NH receives a copy of the BIER packet if the NH cannot be reached due to a link failure. To avoid that the NH distributes further packet copies, the backup F-BM contains only the activated bit for the considered BFER.

We illustrate both computation rules by an example. We consider the BIFT of BFR 1 in Table 5. The backup entry of BFER 6 is an example for the first computation rule. The backup NH for BFR 6 is BFR 4 as it is the NNH of BFR 1 on the path towards BFR 6 in Figure 10. The BFERs reachable from the PLR through BFR 4 are BFER 4 and BFER 6. Therefore, the backup F-BM is `101000`. It can be obtained by bitwise AND'ing the F-BM of BFR 1 for BFER 6 (`111010`) and the F-BM of BFR 2 for BFER 6 (`101100`). The latter can be derived from the multicast subtree of BFR 2 in Figure 10.

The backup entry of BFER 2 is an example for the second computation rule. The backup NH for BFER 2 is BFR 2 and the F-BM contains only one activated bit for BFER 2 (`000010`).

6.3. Properties of BIER-FRR

We have argued that restoration of BIER connectivity may take long time in case of a link failure since this process can start only after the reconvergence of the routing underlay has completed. To shorten the outage time, we introduced BIER-FRR which restores connectivity on the BIER layer as soon as unreachable BFR-NBRs are detected and the connectivity in the routing underlay is restored.

The general concept of BIER-FRR is simple: it requires some sort of detection that a BFR-NBR is no longer reachable, but it does not require any additional signalling as it is a local mechanism. Furthermore, it leverages the restoration of routing underlay so that BIER traffic can profit from FRR mechanisms in the routing underlay. It does not define alternate paths on the BIER layer, which is in contrast to another solution reported in [43].

BIER-FRR comes in two variants: link protection and node protection. Link protection is simple, it just encapsulates BIER traffic into a header of the routing underlay, but it cannot protect against node failures. The encapsulated packet may be sent over an interface over which also a regular copy of the same BIER-packet is transmitted. That means, up to two packet copies can be transmitted over at most one link in case of a failure, which runs in contrast to the actual idea of multicast.

Node protection is more complex. It requires a PLR to send backup copies of a BIER packet to all relevant NNHs encapsulated with a header of the routing underlay. This requires extensions to the BIFT for backup entries. However, it protects against link and node failures. The encapsulated packets may be sent over interfaces over which also a regular copy of the same BIER packet is transmitted. That means that even multiple packet copies can be transmitted over several links in case of a failure.

BIER-FRR is designed for single link and node failures. In case of multiple failures, BIER-FRR suffers from potential shortcomings of the routing underlay to cope with multiple failures, too, so that some traffic may be lost until the BIFT is updated. Furthermore, if both a NH and a NNH fail, the subtree of the NNH is no longer reachable until the BIFTs are updated. Some FRR techniques may cause routing loops in case of multiple failures [12]. In contrast, BIER-FRR cannot cause routing loops because it just leverages the routing underlay and does not propose new paths in failure cases.

6.4. Application of IP-FRR Mechanism on BIER Layer

In Section 3.1 we introduced IP-FRR and described LFAs. In [43] we discussed the application of LFAs on the BIER layer, i.e., in addition to the primary BFR-NBR, the BIFT contains a backup BFR-NBRs respectively, to which a BIER packet is forwarded when the primary NH is unreachable. We identified two major disadvantages. First, their application leaves a significant amount of BFERs unprotected against link or node failures because LFAs cannot guarantee full protection coverage [12]. This holds in particular when node protection is desired for which protection coverage is even lower than for link protection. Second, LFAs on the BIER layer introduce new paths in the BIER topology, which can cause rerouting loops for BIER traffic. Third, this approach assumes IP with IP-FRR as routing underlay while our approach works with any routing underlay and FRR mechanism. Therefore, we argue that the application of IP-FRR mechanisms on BIER layer is not sufficient for appropriate protection.

7. Introduction to P4

This section serves as a primer for readers who are not familiar with P4. First, we explain the general P4 processing pipeline. Then, we describe the concept of match+action tables, control blocks, and metadata. Finally, we explain the recirculate and clone operations.

7.1. P4 Pipeline

P4 is a high-level language for programming protocol-independent packet processors [44]. Its objective is a flexible description of data planes. It introduces the forwarding pipeline shown in Figure 11. A programmable parser reads packets and stores their header information in header fields which are carried together with the packet through the pipeline. The overall processing model is composed of two stages: the ingress and the egress pipeline with a packet buffer

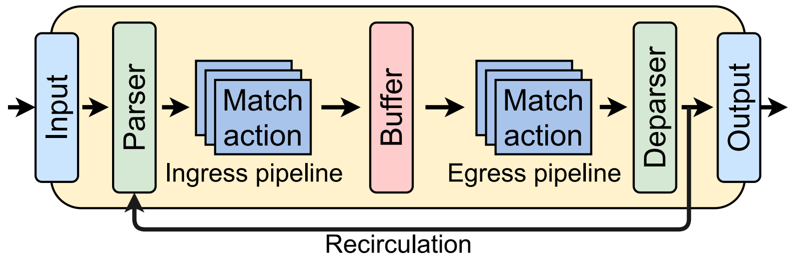


Figure 11: P4 abstract forwarding pipeline according to [44].

in between. The egress port of a packet has to be specified in the ingress pipeline. If no egress port has been specified for a packet at the end of the egress pipeline, the packet is dropped. At the end of the egress pipeline, a deparser constructs the packet with new headers according to the possibly modified header fields. P4 supports the definition and processing of arbitrary headers. Therefore, it is not bound to existing protocols.

7.2. Metadata

Metadata constitute packet-related information. There are standard and user-defined metadata. Examples for standard metadata are ingress port or reception time which are set by the device. User-defined metadata store arbitrary data, e.g., processing flags or calculated values. Each packet carries its own instances of standard and user-define metadata through the P4 processing pipeline.

7.3. Match+Action Tables

Match+action tables are used within the ingress and egress pipeline to apply actions to specified packets. The P4 program describes the structure of each match+action table. The rules are the contents of the table and are added to the table during runtime.

As match+action tables are essential for the description of our prototype, we introduce a compact notation for them by an example. The example is given in Figure 12. The table has the name “MAT_Simple_IP” and describes an implementation of simplified IP forwarding with match+action tables. In the following we use the prefix “MAT_” for naming MATs.

7.3.1. Match Part

A table defines a list of match keys that describe which header fields or metadata are used for matching a packet against the table. The match type indicates the matching method. P4 supports several match types: exact, longest-prefix (lpm), and ternary. The latter features a wildcard match. In our example in Figure 12, the match key is the destination IP address and lpm matching is applied.

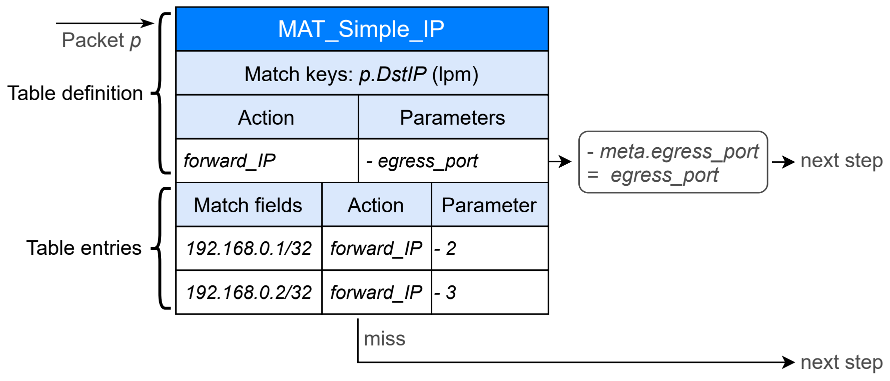


Figure 12: Match+action table for simplified IP forwarding.

7.3.2. Actions

The table further defines a list of actions including their signature which can be used by rules in case of a match. Actions are similar to functions in common programming languages and consist of several primitive operations. Inside an action further actions can be executed. Actions can modify header fields and metadata of a packet. In our example, this is the *forward_IP* action that requires the appropriate egress port as a parameter. Each action is illustrated by a flow chart on the right side of the table.

7.3.3. Rules

During runtime, the match+action tables can be filled with rules through an application programming interface (API). The rules contain match fields which are patterns that are to be matched against a packet's context selected by the match keys. In our example, the match fields are IP addresses. The rules further specify an action in the table definition and suitable parameters which are applied to the packet in case of a match.

In our example in Figure 12 we install two rules. In the first one, the match field is the IP address *192.168.0.1* and it applies the action *forward_IP* with the parameter *2*. This will send packets with the destination IP *192.168.0.1* over port *2*. The match field for the second rule is *192.168.0.2* and it sends the packet over port *3*. For all other destination IPs a miss occurs and no egress port is specified.

When describing match+action tables of our implementation in Section 8, we omit the actual rules as they are configuration data and not part of the P4 implementation.

7.4. Control Blocks

A control block consists of a sequence of match+action tables, operations and if-statements. They encapsulate functionality. Within control blocks other control blocks can be called. Both the ingress and egress pipeline are control blocks that apply other control blocks. We use the prefix "CB_" for naming of our other control blocks. Examples of control blocks in our implementation are *CB_IPv4*, *CB_BIER*, or *CB_Ethernet*.

7.5. Recirculation

P4 does not support native loops. However, as indicated in Figure 11, the recirculation operation returns a packet to the beginning of the ingress pipeline. It activates a standard metadata field, i.e., a flag, which marks the packet for recirculation. The packet still traverses the entire pipeline and only at the end of the egress pipeline the packet is returned to the start of the ingress pipeline. When setting the *recirculate* flag, it is possible to specify which metadata fields should be kept during recirculation. All others are reset to their default values. In contrast, header fields modified during the processing remain modfied after recirculation. Another standard metadata field stores whether a packet has been recirculated.

7.6. Packet Cloning

P4 supports the packet cloning operation clone-ingress-to-egress (*CI2E*). *CI2E* can be called anywhere in the ingress pipeline. This activates the *CI2E* metadata flag which indicates that the packet should be cloned. However, the copy is created only at the end of the ingress pipeline. In the packet clone all header changes are discarded that have been made within the ingress pipeline. If *CI2E* has been called within the ingress pipeline, two packets enter the egress pipeline. One is the original packet that has been processed by the ingress control flow. The second packet is the copy without modifications from the ingress pipeline. Figure 13 illustrates this by an example.

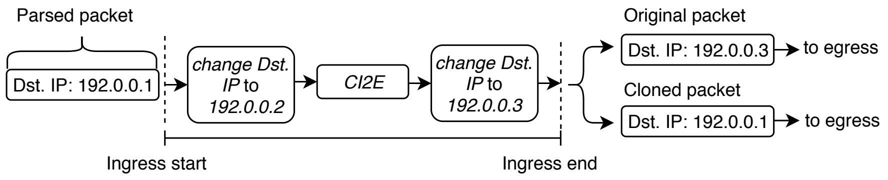


Figure 13: Illustration of the clone-ingress-to-egress (*CI2E*) operation: the destination IP of the clone is the one of the received packet although IP was modified before *CI2E* was called.

When the *CI2E* flag is set, it is possible to specify for the clone whether metadata fields should persist or be reset. When a packet clone enters the egress pipeline, an additional standard metadata flag identifies the packet as a clone. This allows different processing for original and cloned packets.

8. P4-Based Implementation of BIER and BIER-FRR

In this section, we describe the P4-based implementation of IP, IP-FRR, BIER, and BIER-FRR. We first describe the data plane followed by the control plane. In the end, we briefly explain our codebase.

8.1. Data Plane

First, we specify the handling of packet headers, then, we give a high-level overview of the processing pipeline, followed by a detailed explanation of applied control blocks.

8.1.1. Packet Header Processing

P4 requires that potential headers of a packet are defined a priori. Our implementation supports the header suite Ethernet/outer-IP/BIER/inner-IP. We use the inner IP header for regular forwarding and the outer IP header for FRR. During packet processing, headers may be activated or deactivated. Deactivated headers are not added by the deparser. *Encaps* actions in our implementation activate a specific header and set header fields. *Decaps* actions deactivate specific headers.

8.1.2. Overview of Ingress and Egress Control Flow

Figure 14 shows an overview of the entire data plane implementation which is able to perform IP and BIER forwarding as well as IP-FRR and BIER-FRR. It

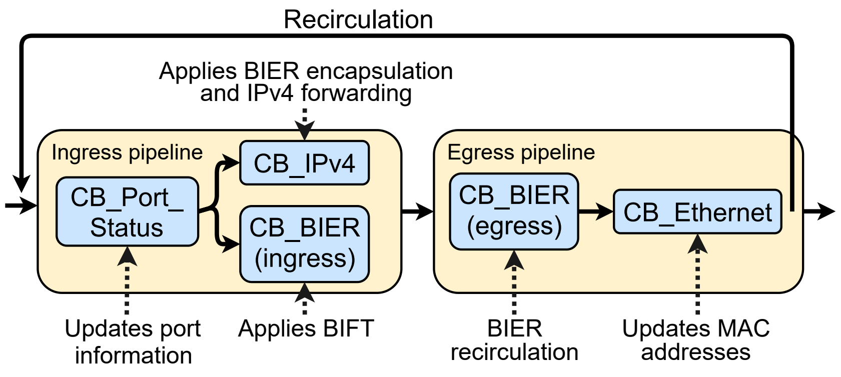


Figure 14: Overview of ingress and egress control flow.

is divided into ingress and egress control flow which are given as control blocks. In the ingress and egress control block the CB_IPv4 and CB_BIER control block are only applied to their respective packets, i.e., the CB_IPv4 control block is applied to IP packets and the CB_BIER control block is applied to BIER packets. We first summarize their operations and describe their implementation in detail in the following Sections.

When a packet enters the ingress pipeline, it is processed by the *CB_Port_Status* control block. It updates the port status (up/down) and records it in the user-defined metadata *meta.live_ports* of the packet. This possibly triggers FRR actions later in the pipeline. Then, the *CB_IPv4* control block or the *CB_BIER* control block is executed depending on the packet type.

The *CB_IPv4* control block is applied to both unicast and multicast IP packets. Unicast packets are processed by setting an appropriate egress port, possibly using IP-FRR in case of a failure. IPMC packets entering the BIER domain are equipped with a BIER header and recirculated for BIER forwarding. IPMC packets leaving the BIER domain are forwarded using native multicast.

The *CB_BIER* control block is applied to BIER packets. There is a *CB_BIER* control block for the ingress control flow and another for the egress control flow. A processing loop for BIER packets is implemented which extends over both *CB_BIER* control blocks. At the beginning of the processing loop in the ingress flow the BitString is copied to metadata *meta.remaining_bits*. This metadata is used to track for which BFERs a copy of the BIER packet still needs to be sent. Then, rules from the *MAT_BIFT* are applied to the packet. This also comprises BIER-FRR actions which encapsulate BIER packets with an IP header if necessary. Within these procedures, the BIER packet is cloned so that the original packet and a clone enter the egress control flow. The processing loop stops if the *meta.remaining_bits* are all zero.

In the *CB_BIER* control block of the egress control flow, the *recirculate* flag is set for cloned packets. At the end of the egress control flow, the clone is recirculated to the ingress control flow with modified *meta.remaining_bits* to continue the processing loop. The non-cloned BIER packet is just passed to the *CB_Ethernet* control block.

The *CB_Ethernet* control block updates the Ethernet header of each packet. Then, the packet is sent if an egress port is set and the *recirculate* flag has not been activated. If the *recirculate* flag is activated, the packet is recirculated instead. This applies to cloned BIER packets in the processing loop or to packets that require a second pass through the pipeline: BIER-encapsulated IPMC packets, BIER-decapsulated IPMC packets, IP-encapsulated BIER packets, or IP-decapsulated BIER packets. If neither *recirculate* flag is activated and nor the egress port is set, the packet is dropped.

8.1.3. CB_Port_Status Control Block

The control block *CB_Port_Status* records whether a port is up or down in the user-defined metadata *meta.live_ports* of a packet. Figure 15 shows that it consists of only the match+action table *MAT_Port_Status*.

The table does not define any match keys. As a result, the first entry matches every packet. We install only a single rule which calls the action *set_port_status*. It copies the parameter *live_ports* to the user-defined metadata *meta.live_ports*. *Meta.live_ports* is a bit string where each bit corresponds to a port of the switch. If the port is currently up, the bit is activated, otherwise, the bit is deactivated. The metadata field *meta.live_ports* is later used by both the *CB_IPv4* and *CB_BIER* control block to decide whether IP-FRR and BIER-FRR should be applied. The parameter *live_ports* in the table is updated by the local controller when the port status changes, which will be explained in Section 8.2.1.

8.1.4. CB_IPv4 Control Block

The *CB_IPv4* control block handles IPv4 packets. Its operation is shown in Figure 16.

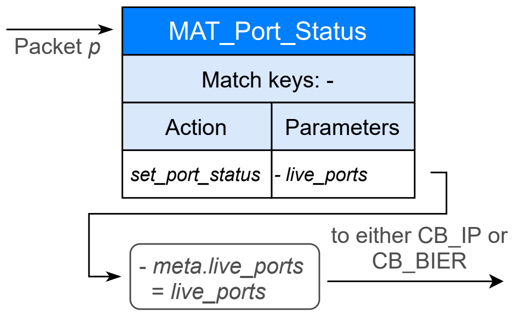


Figure 15: In the control block *CB_Port_Status* the table *MAT_Port_Status* copies the information about live ports to the user-defined metadata field *meta.live_ports* of the packet.

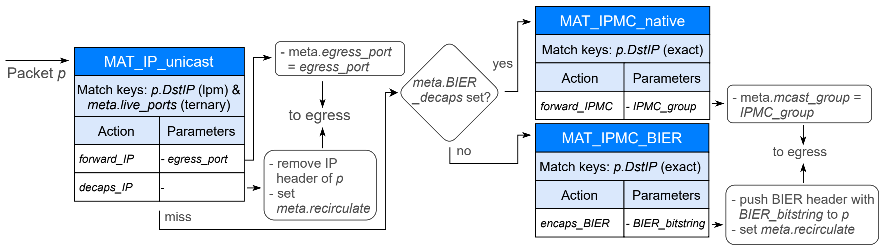


Figure 16: The *CB_IPv4* control block handles IPv4 packets.

It leverages three match+action tables: *MAT_IP_unicast*, *MAT_IPMC_native*, and *MAT_IPMC_BIER*. Packets are processed by these tables depending on their type. *MAT_IP_unicast* performs IP unicast forwarding including IP-FRR. IPMC packets encounter a miss and are relayed by the control flow to *MAT_IPMC_native* or *MAT_IPMC_BIER*. *MAT_IPMC_native* performs native multicast forwarding for IPMC packets leaving the BIER domain while *MAT_IPMC_BIER* just adds a BIER header for IPMC packets entering the BIER domain.

MAT_IP_unicast. This match+action table uses the IP destination address and the metadata *meta.live_ports* as match keys. The IP destination address is associated with a longest prefix match and the *meta.live_ports* with a ternary match. We first explain our implementation of IP-FRR. The rules contain an IP prefix and a *required_port* pattern as match fields (not shown in the table). *Required_port* corresponds to a bit string of all egress ports and is a wildcard expression with only a single zero or one for the primary egress port of the traffic, i.e., `*...*0*...*` or `*...*1*...*`. If FRR is desired for an IP prefix, two rules are provided: a primary rule with `*...*1*...*` as *required_port* pattern, and a backup rule with `*...*0*...*`.

The table offers two actions: *forward_IP* and *decaps_IP*. We explain both in the following in detail.

The *decaps_IP* action is applied to packets that are addressed to the node itself. For such rules the *required_port* pattern is set to `*...*`. Those IP packets are typically BIER packets that have been encapsulated in IP by other nodes for BIER-FRR. Therefore, the IP header is removed and the *recirculate* flag is set so that the packet can be forwarded as BIER packet in a second pass of the pipeline. In theory, other IP packets with the destination IP addresses of the node itself may have reached their final destination. They need to be handed over to a higher layer within the node. However, this feature is not required in our prototype so that we omit it in our implementation.

The *forward_IP* action is applied for other unicast address prefixes and requires an *egress_port* as parameter. It sets the *meta.egress_port* to the indicated egress port so that the packet is switch-internally relayed to the right egress port. The IP-FRR mechanism as explained above may be used in conjunction with *forward_IP* to provide an alternate egress port when the primary egress port is down. This mechanism allows implementation of LFAs.

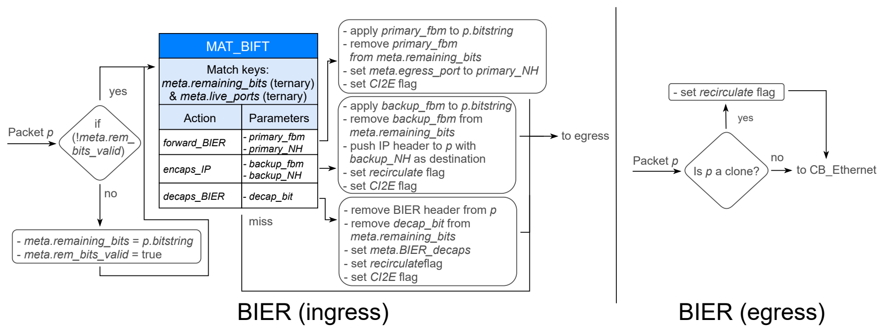
IPMC addresses encounter a miss in this table so that their packets are further treated by the control flow in the *CB_IPv4* control block. It checks whether the *meta.BIER_decaps* bit has been set. If so, the IPMC packet came from the BIER domain and has been decapsulated. Therefore, it is relayed to the *MAT_IPMC_native* table for outbound IPMC traffic. Otherwise, the IMPC packet has been received from a host and requires forwarding through the BIER domain. Therefore, it is relayed to the *MAT_IPMC_BIER* table.

MAT_IPMC_native. This match+action table implements native IPMC forwarding. It is used by a BFER to send IPMC packets to hosts outside the BIER domain that have subscribed to a specific IPMC group. The table *MAT_IPMC_native* uses the IP destination address as match key with an exact match. It defines only the *forward_IPMC* action and requires a switch-internal multicast group as parameter, which is specific to the IPMC group (IP destination address) of the packet. The action sets this parameter in the *meta.mcast_group* of the packet. As a consequence, the packet is processed by the native multicast feature of the switch. This results in packet copies for every egress port contained in the switch-internal multicast group *meta.mcast_group* with the corresponding egress port set in the metadata of the packets. The set of egress ports belonging to that group can be defined through a target-specific interface, which is done by the controller in response to received IGMP packets. Packets encountering a miss in this table are dropped at the end of the pipeline.

MAT_IPMC_BIER. This match+action table uses the IP destination address as match key with an exact match. It defines only the *encaps_BIER* action and requires the bit string as parameter, which is specific to the IPMC group (IP destination address) of the packet. The action pushes a BIER header onto the packet and sets the specified BitString. Then the *recirculate* flag is set so that the packet can be forwarded as a BIER packet in a second pass of the pipeline. Packets encountering a miss in this table are dropped at the end of the pipeline.

8.1.5. CB_BIER Control Block

The *CB_BIER* control block processes BIER packets. It is illustrated in Figure 17.



Packet *p*
if (!*meta.rem_bits_valid*)
yes
no
- *meta.remaining_bits* = *p.bitstring*
- *meta.rem_bits_valid* = true
MAT_BIFT
Match keys: *meta.remaining_bits* (ternary) & *meta.live_ports* (ternary)
Action
Parameters
forward_BIER
- *primary_fbm* - *primary_NH*
encaps_IP
- *backup_fbm* - *backup_NH*
decaps_BIER
- *decap_bit*
miss
- apply *primary_fbm* to *p.bitstring* - remove *primary_fbm* from *meta.remaining_bits* - set *meta.egress_port* to *primary_NH* - set *CI2E* flag
- apply *backup_fbm* to *p.bitstring* - remove *backup_fbm* from *meta.remaining_bits* - push IP header to *p* with *backup_NH* as destination - set *recirculate* flag - set *CI2E* flag
- remove BIER header from *p* - remove *decap_bit* from *meta.remaining_bits* - set *meta.BIER_decaps* - set *recirculate*flag - set *CI2E* flag
to egress
BIER (ingress)
- set *recirculate* flag
yes
Packet *p*
Is *p* a clone?
no
to CB_Ethernet
BIER (egress)

Figure 17: The *CB_BIER* control blocks in the ingress and egress pipeline implement BIER fowarding as a processing loop.

The user-defined metadata *meta.remaining_bits* is used during BIER processing to account for the BFERs that still need a copy of the packet. It serves as a control variable for the processing loop. When a BIER packet is processed by the *CB_BIER* control block for the first time, *meta.remaining_bits* is initialized with the BitString of the packet. The user-defined metadata *meta.remaining_bits_valid* is initially zero. It is activated after *meta.remaining_bits* is initialized and prevents overwriting
meta.remaining_bits when the packet is recirculated.

Then the match+action table *MAT_BIFT* is applied. It implements BIER forwarding including BIER-FRR according to the principle we developed for IP-FRR in Section 8.1.4. Match keys are the packet's *meta.remaining_bits* indicating BFERs, and *meta.live_ports* indicating live egress ports. The match types are ternary. Rules are provided for all individual BFERs both for failure-free cases and failure cases. The match field of these rules consists of two bit strings that we call *dest_BFER* and *required_port* (not shown in the table). The *dest_BFER* bit string has the bit position for the respective BFER activated and all other bit positions set to wildcards (`*...*1*...*`). The *required_port* bit string is used as in Section 8.1.4 to select between primary and backup rules. In case of a match, there are three possible actions.

Decaps_BIER is called by the rule whose activated bit in *dest_BFER* refers to the node itself. It has a F-BM with only the bit of the BFER activated and no primary or backup NH. If this rule matches, the node should receive a copy of the packet. The action removes the BIER header of the packet, activates the user-defined metadata flag *meta.BIER_decaps*, and the *recirculate* flag so that the resulting IPMC packet is processed in a second pass of the pipeline. In addition, the complement of F-BM is used to clear the bit for the processing node itself in *meta.remaining_bits*.

Forward_BIER is called by rules whose activated bit in *dest_BFER* refers to other nodes and where the *required_port* bit string indicates that the egress port works. Thus, *forward_BIER* is used for primary forwarding. It has the primary F-BM and the primary NH (egress port) as parameters. The primary F-BM is applied to clear bits from the BitString of the packet and the complement of the backup F-BM is applied to *meta.remaining_bits*. In addition, *meta.egress_port* is set to the primary NH.

Encaps_IP is called by rules where the *required_port* bit string indicates that the primary egress port does not work for the BFER specified in *dest_BFER*. Thus, *encaps_IP* is used for backup forwarding. It has the backup F-BM and the backup NH (IP address) as parameters. The backup F-BM is applied to clear bits from the BitString of the packet and the complement of the backup F-BM is applied to *meta.remaining_bits*. Then, an IP header is pushed with the destination address of the backup NH. The *recirculate* flag for the packet is activated as it requires IP forwarding in a second run through the pipeline.

At the end of *decaps_BIER*, *forward_BIER*, and *encaps_IP*, a flag for *CI2E* is set. This effects that a packet copy is generated at the end of the ingress pipeline. For the copy (clone), the *recirculate* flag is activated in the *CB_BIER* control block in the egress control flow. With this packet, the BIER processing loop continues. The *meta.remaining_bits* information must be kept to account for the BFERs that still need a packet copy.

When packets enter the *MAT_BIFT* table with *meta.remaining_bits* equal to zero, they encounter a miss. As a result, they are dropped at the end of the pipeline, which stops the processing loop for these BIER packets.

8.1.6. CB_Ethernet Control Block

The *CB_Ethernet* control block is visualized in Figure 18.

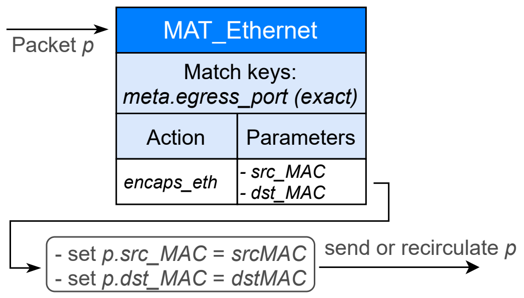


Figure 18: *CB_Ethernet* control block.

It applies the match+action table *MAT_Ethernet* to all packets. The match key is the egress port of the packet and the match type is exact. Only the action *encaps_eth* is defined which requires the parameters *src_MAC* and *dst_MAC*. It updates the Ethernet header of the packet by setting the source and destination MAC address which are provided as parameters. Rules are added for every egress port.

This behavior is sufficient as we assume that any hop is an IP node. Although MAC addresses are not utilized for packet switching, they are still necessary as packet receivers in Mininet discard packets if their destination MAC address does not match their own address.

8.2. Control Plane Architecture

The control plane is visualized in Figure 19. It consists of one global con-

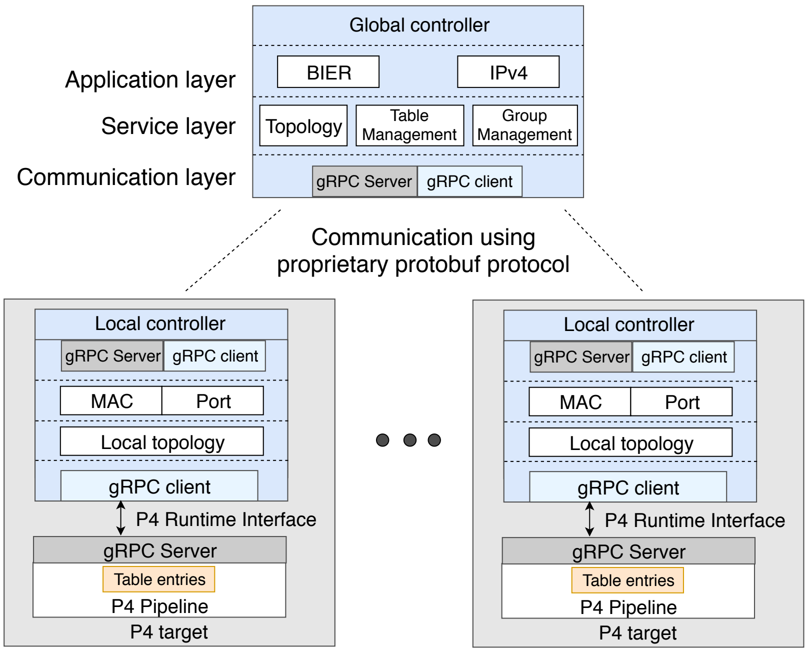


Figure 19: Controller architecture.

troller and one local controller per switch. The local controllers run directly on the switch hardware as P4 switches are mostly whiteboxes. The local controller takes care of tasks that can be performed locally while the global controller is in charge of configuration issues that require a global network view. In theory, a single controller could perform all tasks. However, there are three reasons that call for a local controller: scalability, speed, and robustness. Performing local tasks at the local controller relieves the global controller from unnecessary work. A local controller can reach the switch faster than a global controller. And, most important, a local controller does not need to communicate with the switch via a network. In case of a network failure, the local controller still reaches the switch while the global controller may be unable to do so. Local controllers have also been applied for similar reasons in LoCoSDN [45], P4-MACSec [46], and P4-IPSec [47]. In the following we explain the local and global controller in more detail.

8.2.1. Local Controller

Each switch has a local controller. Switch and local controller communicate via the so-called P4 Runtime which is essentially the Southbound interface in the SDN context. The P4 Runtime uses a gRPC channel and a protobuf-based communication protocol. It allows the controller to write table entries on the switch.

Figure 19 shows that the local controller keeps information about the local topology, learns about neighboring nodes, and port status, and configures this information in the tables of the switch. Moreover, it relays some packets to the global controller and writes table entries as a proxy for the global controller.

We leverage the local controller for three local tasks that we describe in the following: IGMP handling, neighbor discovery, and port monitoring.

IGMP Handling. Multiple hosts are connected to a switch. They leverage the Internet Group Message Protocol (IGMP) to join and leave IPMC groups. If the switch receives an IGMP packet, it forwards it to its local controller which then configures the switch for appropriate actions. For example, it adds a new host to the IPMC group and configures the native IPMC feature of the switch to deliver IPMC packets to the hosts. That feature is used only for carrying multicast traffic from the switch to the hosts. To populate the *MAT_IPMC_native* table, the local controller utilizes the Thrift channel instead of the P4 Runtime as this API is target-specific.

Neighbor Discovery. For neighbor discovery, we implemented a simple proprietary topology recognition protocol. All nodes announce themselves to their neighbors. It allows the local controller to learn the MAC address of the neighbor for each egress port. The local controller stores this information in the match+action table *MAT_Ethernet* which is utilized in the *CB_Ethernet* control block (see Section 8.1.6).

Port Monitoring. A P4 switch by itself is not able to find out whether a neighboring node is reachable. However, a fast indication of this information is crucial to support FRR. In a real network a local controller may test for neighbor reachability, e.g., using a BFD towards all neighbors, loss-of-light, loss-of-carrier, or any other suitable mechanism. Then, the local controller configures this information as a bit string in the match+action table *MAT_Port_Status* of the switch whenever the port status changes. Failure detection is target-dependent and out of scope of this document. Therefore we trigger failure processing of the local controller manually with a software signal. The local controller then activates IP-FRR and BIER-FRR if enabled and notifies the global controller for recomputation of forwarding entries.

8.2.2. Global Controller

We divide the architecture of the global controller in three layers: communication, service, and application (see Figure 19).

The communication layer is responsible for the communication with the local controllers. Each switch is connected to its local controller. Since the P4 runtime only allows one controller with write access, the global controller cannot directly control the switches. Therefore, it communicates with the local controllers to configure the switches. All changes calculated by the global controller are sent to the local controller using a separate channel. The local controller forwards the changes to the switch using the P4 runtime interface.

The service layer provides services for the application layer. This includes information about the topology, multicast groups, and entries in the tables on the switches. The application layer utilizes that information to calculate the table entries.

The global controller receives IGMP messages and keeps track of subscriptions to IPMC groups. If a host is the first to enter or the last to leave an IPMC group at a BFER, the global controller configures the *MAT_IPMC_BIER* table of all BFIRs with an appropriate bit string for the specific IPMC group by activating or deactivating the corresponding bit of the BFER. As a result, the BFIR starts or stops sending traffic from this IPMC group to the BFER.

The global controller sets all entries in the *MAT_IP_unicast* and *MAT_IPMC_BIER* tables of all switches and the entries in the *MAT_BIFT*s. If the global controller is informed by a local controller about a failure, it first reconfigures the *MAT_IP_unicast* and *MAT_IPMC_BIER* tables and then the entries of the *MAT_BIFT*s accordingly.

8.3. Codebase

The implementation of the BIER data plane and control plane including a demo can be downloaded at https://github.com/uni-tue-kn/p4-bier. The provided code contains a more detailed documentation of the BIER(-FRR) implementation. The demo contains several Mininet network topologies that were used to verify the functionality of BIER(-FRR). One of them is described in Section 9.1. Links can be disabled using Mininet, which enables the verification of the BIER-FRR mechanism. A simple host CLI allows multicast packets to be sent and incoming multicast packets to be displayed.

9. Evaluation

In this section we illustrate that BIER traffic is better protected with BIER-FRR. To that end, we conduct experiments in a testbed using our prototype. We first explain the experimental setup, the timing behavior of our emulation and our metrics. Finally, we describe the testbed setup and present experimental protection results in an BIER/IP network with and without IP-FRR and BIER-FRR, for link protection and node protection, respectively.

9.1. Methodology

First, we describe the general approach for our evaluation. Then, we discuss the timing behavior of a software-based evaluation. As the prototype switch is differently controlled than typical routers, we adapt reaction times of the controller after a failure to mimic the timely behaviour of updates for IP forwarding tables and BIFTs. Finally, we explain our metrics.

9.1.1. General Setup

We emulate different topologies in Mininet [48]. The core network is implemented with our P4-based prototype and the software-based *simple_switch* which is based on the BMv2 framework [49]. It forwards IP unicast, IP multicast, and BIER traffic. One source and several subscribers are connected to the core network. The source periodically sends IP unicast and IP multicast packets. IP unicast packets are forwarded as usual through the core network. When IP multicast packets enter the core network, they are encapsulated with a BIER header at the BFIR. BFERs remove BIER headers and forward the IP multicast packets to the subscribers.

Rules for the match+action tables are computed by the global controller in an initial setup phase. In different scenarios we simulate link and node failures and observe packet arrivals at the subscribers. We study different combinations of IP-FRR and BIER-FRR to evaluate the delay until subscribers receive traffic again after a failure has been detected. Also in those cases, the local controller notifies the global controller to perform IP reconvergence and BIFT recomputation because FRR is meant to be only a temporary measure until the global forwarding information base has been updated as a response to the link or node failure.

We report events at the PLR and at all subscribers before and after the failure. For the PLR we show the following signals: failure detection at t_0, updates of IP forwarding entries, and updates of BIFT entries. For the subscribers we record receptions of unicast and multicast packets.

9.1.2. Timing Behavior

Our switch implementation in a small, virtual environment has a different timing behavior than a typical router in a large, physical environment. In particular signaling can be executed with insignificant delay in our virtual environment, e.g., notifying the global controller about the failure or the distribution of updated forwarding entries. This is different with routers and routing protocols in the physical world. Signaling requires significant time as routing protocols need to exchange information about the changed topology. Routers compute alternative routes and push them to their forwarding tables. Only after all unicast paths have been recomputed and globally updated by the routing underlay, BFRs can compute new forwarding entries for BIER and push them to their BIFTs. Thus, the BIFT is updated only significantly later compared to the unicast forwarding information base. To respect that in our evaluation, we configure the global controller to install new IP forwarding entries on the switches only after 150 ms after being informed about a failure and new BIFT entries another 150 ms later.

9.1.3. Metric

We perform experiments with and without IP-FRR and BIER-FRR, and compare the time after which unicast and multicast traffic is delivered again at the subscribers after a failure has been detected by the affected BFR.

9.2. Link Protection

We perform experiments for the evaluation of BIER-FRR with link protection. First, we explain the experimental setup. Afterwards, we report and discuss the results for all scenarios.

9.2.1. Setup for Link Protection

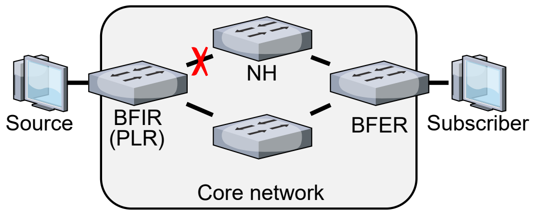


Figure 20: Two hosts the *Source* and the *Subscriber* are connected to a BIER network with IP as the routing underlay.

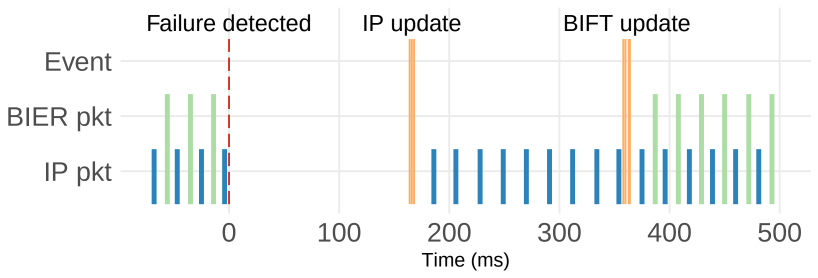
We emulate the testbed depicted in Figure 20 in Mininet. Two hosts the *Source* and the *Subscriber* are connected to a BIER/IP network. The host *Source* sends every 10 ms packets to the host *Subscriber* over the core network. Every other packet is sent by IP unicast and IPMC. The primary path carries packets from *PLR* via *NH* to *BFER*. We simulate the failure of the link between the *PLR* and the *NH* to interrupt packet delivery. We compare the time until the host *Subscriber* receives unicast and multicast traffic again, after the failure has been detected by the *PLR*. We perform experiments with and without IP-FRR and BIER-FRR with link protection.

9.2.2. Without IP-FRR and BIER-FRR

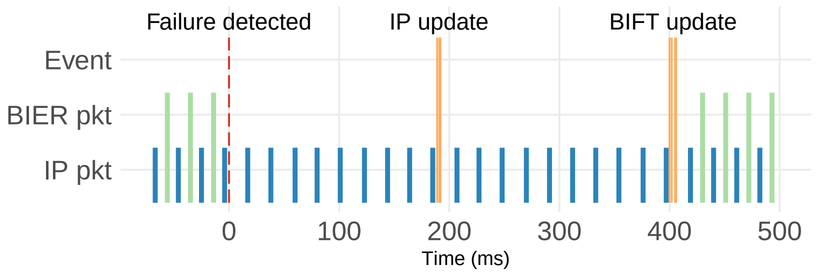
In the first experiment, failure recovery is based only on IP reconvergence and BIFT recomputation. Neither IP-FRR nor BIER-FRR are enabled. Figure 21(a) shows that the failure interrupts packet delivery at the *Subscriber*. Unicast reconvergence is completed after about 170 ms after failure detection. Updating the BIFT entries has finished only after about 370 ms in total. Unicast and multicast packets are received again by the *Subscriber* only after updated IP and BIER forwarding rules from the controller have been installed at the *PLR*.

9.2.3. With IP-FRR but without BIER-FRR

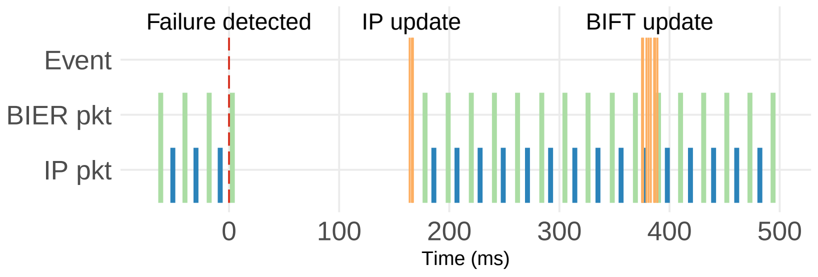
In the second experiment, IP-FRR is enabled but BIER-FRR remains disabled. Figure 21(b) shows that IP unicast traffic immediately benefits from



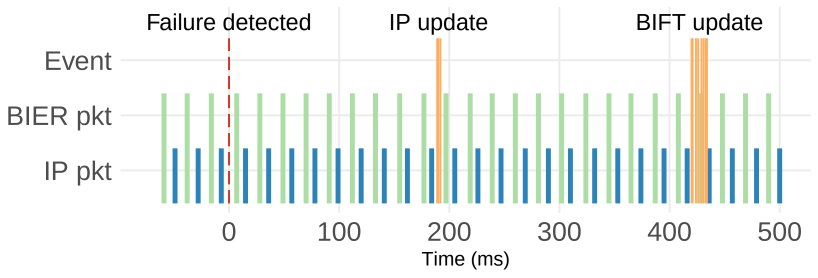
(a) Without IP-FRR and BIER-FRR.



(b) With IP-FRR but without BIER-FRR.



(c) Without IP-FRR but with BIER-FRR.



(d) With IP-FRR and BIER-FRR.

Figure 21: Reception time of packets in the link failure scenario.

IP-FRR when the *PLR* detects the failure. IP-FRR instantly reroutes packets and, therefore, IP unicast traffic is still delivered at the *Subscriber*. Both IP reconvergence and BIFT recompuration are finished slightly later compared to the previous scenario. The reason for the extended duration is that the global controller needs to compute new forwarding entries for IP-FRR during reconvergence, which is not needed if IP-FRR is disabled. After 200 ms, IP reconvergence has finished and the primary IP unicast forwarding entries have been updated. Multicast packets are delivered only after BIFT recomputation after about 400 ms.

9.2.4. Without IP-FRR but with BIER-FRR

In the third experiment, IP-FRR is disabled but BIER-FRR is enabled. Figure 21(c) shows that unicast traffic is delivered at the *Subscriber* when IP reconvergence has finished after about 170 ms. Due to BIER-FRR, BIER traffic benefits from the faster IP reconvergence, too. Multicast traffic is delivered after 170 ms as well, and not only after BIFT recomputation. The BIFT is updated only after about 400 ms in total which is slightly longer than in the scenario without BIER-FRR. Although conceptually the BIFT does not require modification for BIER-FRR with link protection, the match+action tables in the P4 implementation need backup entries that tunnel BIER packets in case of a failure. Therefore, the global controller has to compute new backup entries for BIER-FRR in addition to primary BIFT entries during the recomputation process. The slightly delayed BIFT recomputation is not a disadvantage for BIER traffic because BIER-FRR reroutes BIER packets until both primary and backup BIFT entries have been updated.

9.2.5. With IP-FRR and BIER-FRR

In the last experiment, IP-FRR and BIER-FRR are enabled. Figure 21(d) illustrates that both unicast and multicast traffic are delivered at the *Subscriber* without any delay despite of the failure. This is achieved by FRR mechanisms in both the routing underlay and the BIER layer. IP-FRR immediately restores connectivity for unicast traffic. BIER-FRR leverages the resilient routing underlay to immediately reroute BIER packets. IP reconvergence has finished after about 200 ms. BIFT recomputation finishes only after about 420 ms. In both cases the longer time is explained by the additional FRR entries the global controller has to compute during IP reconvergence and BIFT recomputation, respectively.

9.3. Node Protection

In this paragraph we evaluate BIER-FRR with node protection. First, we describe the experimental setup. Then, we report and discuss the evaluation results for all four scenarios.

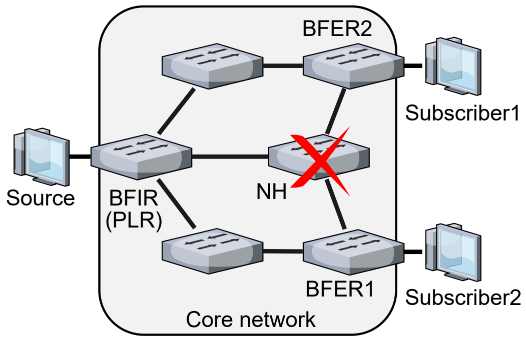


Figure 22: Three hosts, $Source$, $Subscriber1$ and $Subscriber2$ are connected to a BIER network with IP as the routing underlay.

9.3.1. Setup for Node Protection

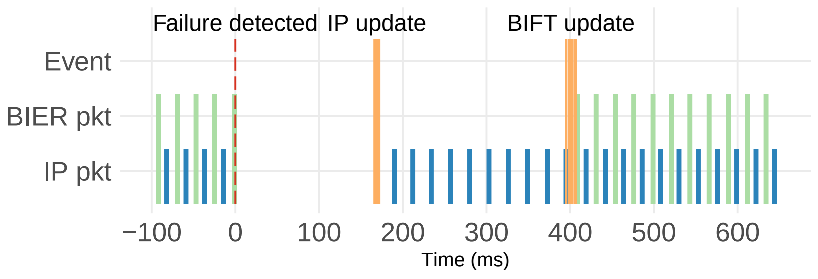
Figure 22 shows the topology we emulated in Mininet. The three hosts $Source$, $Subscriber1$, and $Subscriber2$ are connected to an BIER/IP network. The $Source$ alternately sends two IP unicast packets and one IP multicast packet with 10 ms in between. The unicast packets are sent to $Subscriber1$ and $Subscriber2$. The IPMC group of the the IPMC packet is subscribed by $Subscriber1$ and $Subscriber2$. On the primary path, packets are carried from the PLR via the NH to $BFER1$ and $BFER2$, respectively. We simulate the failure of the NH to interrupt packet delivery with a node failure. We evaluate the time until both the $Subscriber1$ and the $Subscriber2$ receive traffic again after the PLR detects the failure. We perform experiments with and without IP-FRR and BIER-FRR with node protection. We discuss the outcome and show figures only for $Subscriber1$ because results for $Subscriber2$ are very similar.

9.3.2. Without IP-FRR and BIER-FRR

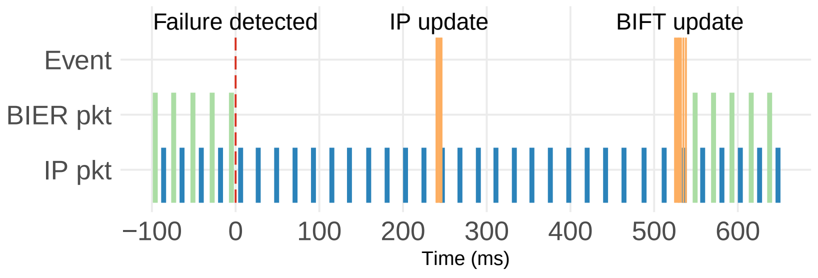
In the first scenario, the local controller at the PLR triggers only IP reconvergence and BIFT recomputation after failure detection. No FRR measures are enabled. Figure 23(a) shows that the $Subscriber1$ receives IP unicast traffic only after IP reconvergence which takes about 180 ms. $Subscriber1$ receives multicast traffic only after BIFT recomputation which takes about 400 ms. Both IP reconvergence and BIFT recomputation require slightly more time than in the link failure scenario because now the local controller reports a node failure which requires more rules to be recomputed.

9.3.3. With IP-FRR but without BIER-FRR

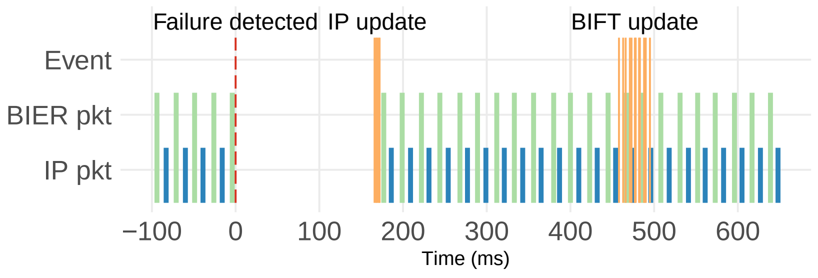
In the second scenario, IP-FRR is enabled but not BIER-FRR. Figure 23(b) shows that IP unicast traffic immediately benefits from IP-FRR. Traffic is delivered at the $Subscriber1$ without any delay despite of the failure. IP reconvergence requires about 240 ms. Multicast traffic is received by the $Subscriber1$ only after BIFT recomputation which has finished only after about 520 ms.



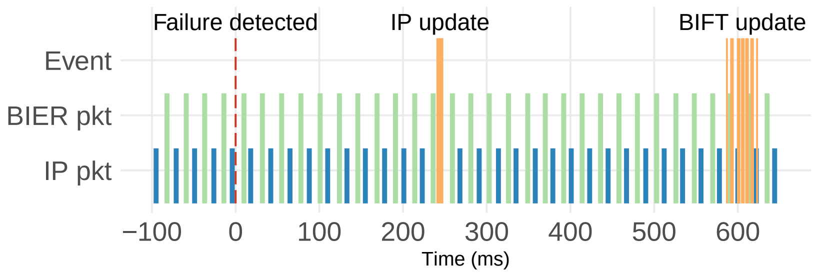
(a) Without IP-FRR and BIER-FRR.



(b) With IP-FRR but without BIER-FRR.



(c) Without IP-FRR but with BIER-FRR.



(d) With IP-FRR and BIER-FRR.

Figure 23: Reception time of packets in the node failure scenario.

Again, IP reconvergence and BIFT recomputation require slightly more time than without IP-FRR because because additional IP-FRR entries have to be computed.

9.3.4. Without IP-FRR but with BIER-FRR

In the third scenario, BIER-FRR is enabled but not IP-FRR. Figure 23(b) shows that both IP unicast and multicast traffic are received at the $Subscriber1$ only after IP reconvergence which takes about 170 ms. Afterwards, IP traffic is rerouted because of the updated forwarding entries. BIER traffic is rerouted after that time as well, because BIER-FRR leverages the updated routing underlay instead of requiring BIFT recomputation which has finished only after about 500 ms.

9.3.5. With IP-FRR and BIER-FRR

In the last scenario, both IP-FRR and BIER-FRR are enabled. Figure 23(d) shows that both IP unicast and multicast traffic are received by the $Subscriber1$ without any delay despite of the failure. IP-FRR reroutes IP unicast traffic as soon as the failure is detected by the PLR. Similarly, BIER-FRR reroutes BIER traffic immediately, too. Therefore, BIER traffic benefits from the resilience of the routing underlay to forward BIER traffic although the NH failed and BIFT recomputation has not finished, yet. IP reconvergence takes about 240 ms. BIFT recomputation finished only after 600 ms.

10. Conclusion

BIER is a novel, domain-based, scalable multicast transport mechanism for IP networks that does not require state per IP multicast (IPMC) group in core nodes. Only ingress nodes of a BIER domain maintain group-specific information and push a BIER header on multicast traffic for simplified forwarding within the BIER domain. Bit-forwarding routers (BFRs) leverage a bit index forwarding table (BIFT) for forwarding decisions. Its entries are derived from the interior gateway protocol (IGP), the so-called routing underlay. In case of a failure, the BIFT entries are recomputed only after IP reconvergence. Therefore, BIER traffic encounters rather long outages after link or node failures and cannot profit from fast reroute (FRR) mechanisms in the IP routing underlay.

In this work, we proposed BIER-FRR to shorten the time until BIER traffic is delivered again after a failure. BIER-FRR deviates BIER traffic around the failure via unicast tunnels through the routing underlay. Therefore, BIER benefits from fast reconvergence or FRR mechanisms of the routing underlay to deliver BIER traffic as soon as connectivity for unicast traffic has been restored in the routing underlay. BIER-FRR has a link and a node protection mode. Link protection is simple but cannot protect against node failures. To that end, BIER-FRR offers a node protection mode which requires extensions to the BIFT structure.

As BIER defines new headers and forwarding behavior, it cannot be configured on standard networking gears. Therefore, a second contribution of this paper is a prototype implementation of BIER and BIER-FRR on a P4-programmable switch based on P4$_{16}$. It works without extern functions or other extensions such as local agents that impede portability. The switch offers an API for interaction with controllers. A local controller takes care of local tasks such as MAC learning and failure detection. A global controller configures other match+action tables that pertain to forwarding decisions. A predecessor of this prototype without BIER-FRR and based on P4$_{14}$ has been presented as a demo in [5]. The novel BIER prototype including BIER-FRR demonstrates that P4 facilitates implementation of rather complex forwarding behavior.

We deployed our prototype on a virtualized testbed based on Mininet and the software switch BMv2. Our experiments confirm that BIER-FRR significantly reduces the time until multicast traffic is received again by subscribers after link or node failures. Without BIER-FRR, multicast packets arrive at the subscriber only after reconvergence of the routing underlay and BIFT recomputation. With BIER-FRR, multicast traffic is delivered again as soon as connectivity in the routing underlay is restored, which is particularly fast if the routing underlay applies FRR methods.

Acknowledgment

The authors thank Wolfgang Braun and Toerless Eckert for valuable input and stimulating discussions.

References

[1] D. Merling, M. Menth, et al., An Overview of Bit Index Explicit Replication (BIER), IETFJournal (Mar. 2018).

[2] I. Wijnands, E. Rosen, et al., RFC8279: Multicast Using Bit Index Explicit Replication (BIER), https://tools.ietf.org/html/rfc8279 (Nov. 2017).

[3] M. Shand, S. Bryant, IP Fast Reroute Framework, https://tools.ietf.org/html/rfc5714 (Jan. 2010).

[4] D. Merling, M. Menth, BIER Fast Reroute, https://datatracker.ietf.org/doc/draft-merling-bier-frr/ (Mar. 2019).

[5] W. Braun, J. Hartmann, et al., Demo: Scalable and Reliable Software-Defined Multicast with BIER and P4, IFIP/IEEE International Symposium on Integrated Network Management (IM) (May 2017).

[6] The P4 Language Consortium, The P4 Language Specification Version 1.0.5, https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf (Nov. 2018).

[7] The P4 Language Consortium, The P4 Language Specification Version 1.1.0, https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf (Nov. 2018).

[8] H. Holbrook, B. Cain, et al., Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast, https://tools.ietf.org/html/rfc4604 (Aug. 2006).

[9] T. Speakman, J. Crowcroft, et al., PGM Reliable Transport Protocol Specification, https://tools.ietf.org/html/rfc3208 (Dec. 2001).

[10] G. Rétvári, J. Tapolcai, et al., IP fast ReRoute: Loop Free Alternates revisited, IEEE Conference on Computer Communications (Apr. 2011).

[11] D. Katz, D. Ward, et al., Bidirectional Forwarding Detection (BFD), https://tools.ietf.org/html/rfc5880 (Jun. 2010).

[12] D. Merling, W. Braun, et al., Efficient Data Plane Protection for SDN, IEEE Conference on Network Softwarization and Workshops (Jun. 2018).

[13] M. Menth, M. Hartmann, et al., Loop-Free Alternates and Not-Via Addresses: A Proper Combination for IP Fast Reroute?, Computer Networks 54 (Jun. 2010).

[14] A. Raj, O. Ibe, et al., A survey of IP and multiprotocol label switchingfast reroute schemes, Computer Networks 51 (Jun. 2007).

[15] V. S. Pal, Y. R. Devi, et al., A Survey on IP Fast Rerouting Schemes using Backup Topology, International Journal of Advanced Research inComputer Science and Software Engineering 3 (Apr. 2003).

[16] S. Bryant, S. Previdi, et al., A Framework for IP and MPLS Fast Reroute Using Not-Via Addresses, https://tools.ietf.org/html/rfc6981 (Aug. 2013).

[17] L. Csikor, G. Rétvári, et al., IP fast reroute with remote Loop-Free Alternates: The unit link cost case, International Congress on Ultra Modern Telecommunications and Control Systems (Feb. 2012).

[18] S. Islam, N. Muslim, et al., A Survey on Multicasting in Software-Defined Networking, IEEE Communications Surveys Tutorials 20 (Nov. 2018).

[19] Z. Al-Saeed, I. Ahmada, et al., Multicasting in Software Defined Networks: A Comprehensive Survey, Journal of Network and Computer Applications 104 (Feb. 2018).

[20] J. Rückert, J. Blendin, et al., Software-Defined Multicast for Over-the-Top and Overlay-based Live Streaming in ISP Networks, Journal of Network and Systems Management 23 (Apr. 2015).

[21] J. Rückert, J. Blendin, et al., Flexible, Efficient, and Scalable Software-Defined Over-the-Top Multicast for ISP Environments With DynSdm, IEEE Transactions on Network and Service Management 13 (Sep. 2016).

[22] L. H. Huang, H.-J. Hung, et al., Scalable and Bandwidth-Efficient Multicast for Software-Defined Networks, IEEE Global Communications Conference (Dec. 2014).

[23] S. Zhou, H. Wang, et al., Cost-Efficient and Scalable Multicast Tree in Software Defined Networking, Algorithms and Architectures for Parallel Processing (Dec. 2015).

[24] J.-R. Jiang, S.-Y. Chen, Constructing Multiple Steiner Trees for Software-Defined Networking Multicast, Proceedings of the 11th International Conference on Future Internet Technologies (Jun. 2016).

[25] Y.-D. Lin, Y.-C. Lai, et al., Scalable Multicasting with Multiple Shared Trees in Software Defined Networking, Journal of Network and Computer Applications 78 (Jan. 2017).

[26] Z. Hu, D. Guo, et al., Multicast Routing with Uncertain Sources in Software-Defined Network, IEEE/ACM International Symposium on Quality of Service (Jun. 2016).

[27] B. Ren, D. Guo, et al., The Packing Problem of Uncertain Multicasts, Concurrency and Computation: Practice and Experience 29 (August 2017).

[28] A. Iyer, P. Kumar, et al., Avalanche: Data Center Multicast using Software Defined Networking, International Conference on Communication Systems and Networks (Jan 2014).

[29] W. Cui, C. Qian, et al., Scalable and Load-Balanced Data Center Multicast, IEEE Global Communications Conference (Dec 2015).

[30] S. H. Shen, L.-H. Huang, et al., Reliable Multicast Routing for Software-Defined Networks, IEEE Conference on Computer Communications (April 2015).

[31] M. Popovic, R. Khalili, et al., Performance Comparison of Node-Redundant Multicast Distribution Trees in SDN Networks, International Conference on Networked Systems (Apr. 2017).

[32] T. Humernbrum, B. Hagedorn, et al., Towards Efficient Multicast Communication in Software-Defined Networks, IEEE International Conference on Distributed Computing Systems Workshops (Jun. 2016).

[33] D. Kotani, K. Suzuki, et al., A Multicast Tree Management Method Supporting Fast Failure Recovery and Dynamic Group Membership Changes in OpenFlow Networks, Journal of Information Processing 24 (2016).

[34] T. Pfeiffenberger, J. L. Du, et al., Reliable and Flexible Communications for Power Systems: Fault-tolerant Multicast with SDN/OpenFlow, International Conference on New Technologies, Mobility and Security (Jul. 2015).

[35] W. K. Jia, L.-C. Wang, et al., A Unified Unicast and Multicast Routing and Forwarding Algorithm for Software-Defined Datacenter Networks, IEEE Journal on Selected Areas in Communications 31 (Dec. 2013).

[36] M. J. Reed, M. Al-Naday, et al., Stateless Multicast Switching in Software Defined Networks, IEEE International Conference on Communications (May 2016).

[37] A. Giorgetti, A. Sgambelluri, et al., First Demonstration of SDN-based Bit Index Explicit Replication (BIER) Multicasting, European Conference on Networks and Communications (Jun. 2017).

[38] A. Giorgetti, A. Sgambelluri, et al., Bit Index Explicit Replication (BIER) Multicasting in Transport Networks, International Conference on Optical Network Design and Modeling (May 2017).

[39] T. Eckert, G. Cauchie, et al., Traffic Engineering for Bit Index Explicit Replication BIER-TE, http://tools.ietf.org/html/draft-eckert-bier-te-arch (Nov. 2017).

[40] W. Braun, M. Albert, et al., Performance Comparison of Resilience Mechanisms for Stateless Multicast Using BIER, IFIP/IEEE International Symposium on Integrated Network Management (May 2017).

[41] Q. Xiong, G. Mirsky, et al., The Resilience for BIER, https://datatracker.ietf.org/doc/draft-xiong-bier-resilience/ (Mar. 2019).

[42] Q. Xiong, G. Mirsky, et al., BIER BFD, https://datatracker.ietf.org/doc/draft-hu-bier-bfd/ (Mar. 2019).

[43] D. Merling, S. Lindner, et al., Comparison of Fast-Reroute Mechanisms for BIER-Based IP Multicast, International Conference on Software Defined Systems (Apr. 2020).

[44] P. Bosshart, D. Daly, et al., P4: Programming Protocol-Independent Packet Processors, ACM SIGCOMM Computer Communication Review 44 (Jul. 2014).

[45] M. Schmidt, F. Hauser, et al., LoCoSDN: A Local Controller for Operation of OFSwitches in non-SDN Networks, Software Defined System (Apr. 2018).

[46] F. Hauser, M. Schmidt, et al., P4-MACsec: Dynamic Topology Monitoring and Data Layer Protection with MACsec in P4-Based SDN, IEEE Access (Mar. 2020).

[47] F. Hauser, M. Häberle, et al., P4-IPsec: Implementation of IPsec Gateways in P4 with SDN Control for Host-to-Site Scenarios, ArXiv (Jul. 2019).

[48] B. Lantz, B. Heller, et al., A Network in a Laptop: Rapid Prototyping for Software-defined Networks, ACM SIGCOMM HotNets Workshop (Oct. 2010).

[49] p4lang, behavioral-model, `https://github.com/p4lang/behavioral-model` (Mar. 2019).

2.4 RAP Extensions for the Hybrid Configuration Model

RAP Extensions for the Hybrid Configuration Model

Lukas Osswald[*], Steffen Lindner[*], Lukas Wüsteney[†], and Michael Menth[*]
[*]University of Tuebingen, Chair of Communication Networks, 72076 Tuebingen, Germany
[†]Hirschmann Automation and Control GmbH, CTO Office, 72654 Neckartenzlingen, Germany
Email: {lukas.osswald,steffen.lindner, menth}@uni-tuebingen.de, lukas.wuesteney@belden.com

***Abstract*—Modern applications in industrial automation rely on a deterministic network service, i.e., low latency, high reliability, and network convergence. Therefore, the IEEE 802.1 TSN Task Group introduces Time-Sensitive Networking (TSN). Besides mechanisms for traffic shaping, time synchronization, and reliability, TSN introduces three different configuration models for resource reservation: the fully distributed, the fully centralized, and the centralized network/distributed user model. Furthermore, IEEE P802.1Qdd specifies the Resource Allocation Protocol (RAP) to enable resource reservation for TSN streams in the fully distributed model. In this paper we give an introduction to RAP, and propose extensions to RAP for use in the hybrid configuration model. Additionally, we implement a prototype which is published under an open-source license.**

***Index Terms*—TSN, Resource Allocation Protocol (RAP), 802.1Qdd, centralized network/distributed user model, CUC**

I. Introduction

Many modern industrial applications, e.g., automation, require an ultra-low latency, deterministic network service. Typically, networks provide such a high quality of service (QoS) by reserving bandwidths for flows using resource reservation protocols. Audio/Video Bridging (AVB) is a standard for realtime communication in Ethernets. It is further developed under the name Time-Sensitive Networking (TSN) to meet even stricter time constraints. The IEEE 802.1 TSN Task Group (TG) defines concepts and protocols for resource management, time synchronization, bounded latency and delay variation, avoidance of congestion-based packet loss, as well as for reliability. The objectives are deterministic services for unidirectional unicast and multicast streams which carry realtime data. Thus, TSN supports the transport of multiple protocols for realtime applications over the same link, and facilitates integration of IT and Operational Technology (OT) networks.

AVB introduced the Stream Reservation Protocol (SRP) for admission control of streams. It is a hop-by-hop reservation protocol with local resource management carried out by every node, and supports a distributed configuration model. IEEE Std 802.1Qcc [7] extends SRP to support additional TSN features. Additionally, two new central entities for centralized network management are introduced. An application-specific Centralized User Configuration (CUC) receives flow requests including requirements from applications, communicates them in a uniform way to the Centralized Network Configuration (CNC) which is responsible for resource management and configures the switches to treat the flows with the requested QoS. In addition, IEEE Std 802.1Qcc introduces the centralized network/distributed user model, we further refer to as the hybrid configuration model. It leverages the CNC and end station convey stream requirements with a distributed resource reservation protocol to the network. The first bride, connected to an end stations, directly forwards the request to the CNC instead of performing distributed signalling. TSN offers new mechanisms to provide QoS. The Resource Allocation Protocol (RAP) is being defined by the TSN TG in IEEE P802.1Qdd to support these new mechanisms. RAP is a hop-by-hop protocol for dynamic resource reservation based on the Link-local Registration Protocol (LRP) for transport purposes.

The contribution of this paper is manifold. We survey RAP and LRP, and point out the improvements to SRP. As IEEE Std 802.1Qcc introduces centralized resource We analyse RAP's data model and propose extensions for RAP to be able to reserve scheduled streams in a centrally managed TSN network. Therefore, we describe a network based on the centralized network/distributed user model which includes a CNC, a new central component, called RAP-CUC, and leverages RAP for resource reservation of streams. Then, we provide a detailed architecture for a general CUC component, whose core is independent of a user-specific protocol and CNC implementation. Following that, we derive a specific CUC for RAP as a user-specific protocol.

II. Resource Reservation and QoS Mechanisms

In this section we give an overview of resource reservation and quality of service (QoS) mechanisms in realtime networks, in particular in Audio/Video Bridging (AVB) and Time-Sensitive Networking (TSN) Ethernet. Further, we give an introduction to the resource reservation protocols of AVB.

A. *Resource Reservation in Realtime Networks*

Realtime streams have QoS requirements, e.g., bounded delay and delay variation, and minimum throughput. Bridges apply special mechanisms to guarantee the QoS for such streams despite of other traffic load. For instance, they limit the amount of realtime traffic to avoid overload in the network. This is performed per stream and/or per aggregate with the

This work has been supported by the German Federal Ministry of Education and Research (BMBF) under support code 16KIS1161 (Collaborative Project KITOS). The authors alone are responsible for the content of the paper.

help of shapers or policers. Another option is to assign dedicated transmission slots to frames, which is also known as scheduled traffic.

In both cases, admission control is performed. That is, realtime streams are explicitly admitted. Their properties, e.g., transmission rate and burst size, are considered for bookkeeping. Additional streams are admitted only if the remaining transmission capacity suffices. Otherwise, admission requests are declined to protect the QoS of already admitted streams. For this purpose, static and dynamic resource reservation exist which we review in the following.

1) Static Resource Reservation: With static resource reservation, the realtime streams and their properties are known prior to computation of configuration. Only a supportable amount of traffic is admitted and configured to obtain preferential treatment by the network. Other streams cannot demand this level of QoS at runtime.

2) Dynamic Resource Reservation: Dynamic resource reservation leverages network protocols to signal admission requests, configure QoS mechanisms in switches, and inform the requesting entity about the result. A resource reservation protocol conveys the properties of streams such that the network can take admission decisions. Such a resource reservation protocol can be classified as distributed or agent-based.

Distributed resource reservation leverages a resource reservation protocol which signals stream requirements along the path of the relevant stream. In case of success, it applies configurations to the bridges along the path of the admitted stream.

Agent-based resource reservation utilizes a centralized controller with a global view. End stations communicate stream properties and QoS requirements to the centralized controller which computes configuration data and applies them to bridges and hosts along the path of the stream.

In AVB networks, dynamic, distributed resource reservation is used. In TSN networks, both dynamic and static resource reservation are supported.

B. Resource Reservation and Traffic Shaping in AVB

In AVB [3], senders and receivers of a stream are denoted as Talkers and Listeners. Subsequently, we give a short introduction to resource reservation and traffic shaping in AVB.

1) Resource Reservation: The Stream Reservation Protocol (SRP) supports dynamic, distributed resource reservation in AVB networks. SRP leverages three protocols for resource reservation: the Multiple VLAN Registration Protocol (MVRP), the Multiple MAC Registration Protocol (MMRP), and the Multiple Stream Registration Protocol (MSRP). These three protocols use the Multiple Registration Protocol (MRP) as transport layer. MRP transmits the data provided by application protocols through the network where it is persistently stored in each hop [6].

VLANs are used to limit the scope of streams within the network. End stations may join a VLAN using MVRP. With MMRP a station subscribes to traffic from specific multicast or unicast MAC addresses. As a result, forwarding rules are configured in the briges along the path from the Talker to the Listener within the corresponding VLAN. MSRP is used to reserve resources for these streams.

IEEE Std 802.1Qcc extends the capability of SRP for TSN by introducing a central control elements for network management and users control. We discuss the configuration models of TSN in Section III.

2) Traffic Shaping: AVB introduces two traffic classes: Class A and Class B. A maximum delay of 2 ms and 50 ms is guaranteed for the traffic of the respective traffic classes over up to seven hops [3]. To achieve that, the traffic of both classes is policed with a Credit-Based Shaper (CBS) [2] which is a token bucket based algorithm to limit the burst size and bandwidth of traffic aggregates.

C. Selected QoS Functions in TSN

The TSN standards specify multiple mechanisms to guarantee QoS for realtime streams. Some of them are adopted from AVB, others are new [8], [18]. In the following, we describe two TSN-specifc mechanisms that are relevant in the context of this paper.

1) Time-Aware Shaper (TAS): To achieve ultra-low latency and delay variation for applications requiring hard realtime, TSN supports scheduled traffic. The Time-Aware Shaper (TAS) [4] leverages a TDMA paradigm. All bridges are synchronized in time and forward traffic according to a global schedule without queuing delay. If end stations are not synchronized, their traffic may be buffered at the access bridge. Optionally, Talkers which are synchronized to the network may be included in the schedule.

TAS is optimally applicable with static and dynamic, agent-based resource reservation. With agent-based resource reservation, time-aware Talkers communicate the earliest and latest possible transmit time of each stream to the network. After schedule synthesis, the network notifies the Talkers about the precise start of transmit in an interval for each stream.

2) Frame Replication and Elimination for Reliability (FRER): IEEE Std 802.1CB introduces Frame Replication and Elimination for Reliability (FRER) [5] to TSN which enables seamless redundancy over multiple paths for a stream. Frames are replicated at a bridge and sequence numbers are attached. The duplicated frames are sent along disjoint paths to another bridge which eliminates duplicate frames with the help of the sequence numbers. If one path fails, the traffic still reaches the destination over the working path.

III. Configuration Models for Resource Reservation in TSN

This section presents the configuration models for TSN specified in IEEE Std 802.1Qcc [7]. First, we give an overview of the User/Network Interface (UNI). Afterwards, we examine the configuration models, i.e., the fully centralized, the fully distributed, and the centralized network/distributed user model, further referred to as the hybrid configuration model.

A. User/Network Interface (UNI)

IEEE Std 802.1Qcc defines a User/Network Interface (UNI) leveraging YANG as a modeling language. The UNI is a bidirectional interface that is used to communicate QoS requirements and stream properties of end stations, and propagate the admission control status of a stream from the network to the end stations. For that, it consists of four YANG groupings that are further specified below.

1) 802.1Qcc Group-Talker: The group-talker is used to convey the Talker related stream properties, QoS requirements, and TSN capabilities for a stream to the network. It contains fields for specifying traffic characteristics of the stream, i.e., interval, maximum frame size and maximum amount of frames per interval. Time-aware Talkers additionally communicate the earliest and the latest possible transmission start time within an interval. Further, QoS properties like maximum latency and delay variation can be included as user-to-network requirements. A stream rank is included which is used by bridges to determine streams to be dropped in an oversubscription scenario. Additionally, the Talker discloses the TSN capabilities of its interface and specifies the characteristics of a stream's frame such that the network is able to associate it with its stream.

2) 802.1Qcc Group-Listener: The group-listener is intended as an admission control request for Listeners to indicate participation in a stream. It comprises QoS requirements of a Listener and its TSN capabilities.

3) 802.1Qcc Group-Status-Stream: The group-status-stream defines information about the admission control status of streams. The included information originates from the network. The grouping includes a status code for the Talkers and the Listeners each. In case of a failure, additional failure information specifies the cause and identifies the device by MAC address and interface name.

4) 802.1Qcc Group-Status-Talker-Listener: The group-status-talker-listener comprises status information and configuration data for one Talker or Listener. It originates from the network as a result of admission control procedure. It includes the worst-case latency a frame of a stream can experience along its path. Additionally, configuration data is provided to the end station, e.g., the point in time a time-aware Talker has to start transmission, or the VLAN ID and priority used for stream identification.

B. Fully Centralized Configuration Model

Figure 1 illustrates the fully centralized configuration model. The fully centralized model introduces a central network management controller, i.e., Centralized Network Configuration, and one or more controllers for user management, i.e., Centralized User Configuration.

(1) End stations signal admission control requests to a Centralized User Configuration (CUC) via a user-specific protocol. A commonly used user-specific protocol is OPC-UA client-server, a protocol developed by the OPC Foundation [1].

(2) The CUC collects stream requirements of all users participating in the same stream.

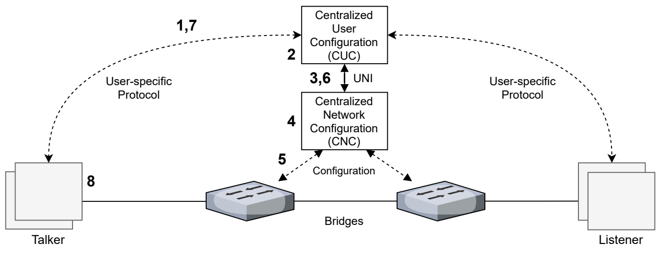


Figure 1: The fully centralized configuration model is composed of end stations, bridges, one or more CUCs, and a single CNC. The CNC-CUC interface is defined by the User/Network Interface (UNI)

(3) When all users provide sufficient information, the CUC initiates the resource reservation process via the UNI with the Centralized Network Configuration (CNC).

(4) A single CNC provides central network management. It takes admission control decisions based on the users' requirements obtained from all CUCs. The CNC computes configurations for bridges and end stations.

(5) It configures the bridges with network management protocols like SNMP [10], RESTCONF [9], or NETCONF [13].

(6) The CNC communicates the computed configuration for end stations and admission control status to the CUC.

(7) The CUC propagates both to the respective end stations.

(8) End stations reconfigure their interfaces accordingly and can start data transmission. Afterwards, the CNC and CUC can dynamically react on events, like node/link failures.

IEEE P802.1Qdj [16] further enhances the definition of the interface between CNC-CUC. Therefore, it will provide a fully functional YANG model for the centralized configuration model in TSN. This model will based on the four YANG groupings of the current UNI and may add extensions. As a consequence, the implementation of a CNC-CUC interface can be based on a YANG based protocol, e.g. RESTCONF or NETCONF.

The fully centralized model introduces network management with a global view to TSN networks. This enables computation of globally optimized schedules for TAS, eliminating queuing delay [12]. This makes the fully centralized model suitable for environments that require precise timing of packets and complex planning.

C. Fully Distributed Configuration Model

Figure 2 illustrates the fully distributed configuration model. The fully distributed configuration model performs admission control decisions with a hop-by-hop resource reservation protocol. As a consequence, the UNI is located between all participating devices. SRP may be used for that purpose. However, the Resource Allocation Protocol (RAP) is currently defined to provide support for novel TSN mechanisms. End stations signal stream requirements to the network. The bridges take admission control decisions based on local information

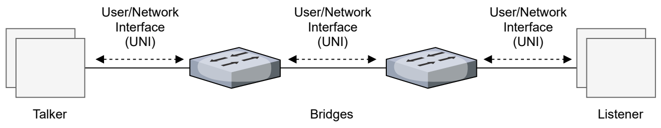

User/Network Interface (UNI)
User/Network Interface (UNI)
User/Network Interface (UNI)
Talker
Bridges
Listener

Figure 2: The fully distributed configuration model is composed of end stations and bridges and was originally defined for AVB networks.

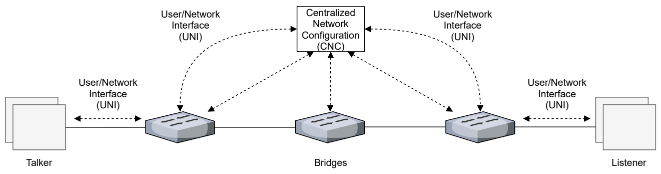

User/Network Interface (UNI)
Centralized Network Configuration (CNC)
User/Network Interface (UNI)
User/Network Interface (UNI)
User/Network Interface (UNI)
Talker
Bridges
Listener

Figure 3: The centralized network/distributed user model is composed of end stations, bridges and a single CNC.

only. Therefore, computation of a globally optimized schedule for TAS is not optimally possible.

D. Hybrid Configuration Model

Figure 3 illustrates the hybrid configuration model. End stations communicate admission control requests via a distributed resource reservation protocol. The edge bridges ensure that admission control requests are directly forwarded to the CNC, and not hop-by-hop as in the fully distributed model. The CNC then takes admission control decisions and computes configurations as in the fully centralized model. As a consequence, the UNI is located between end stations, the edge bridge, and the CNC.

The hybrid model avoids the use of multiple, application specific CUCs by relying on a single protocol for resource reservation. This protocol can provide admission control as a service to multiple applications. This can reduce implementation complexity of end stations and the network management components. Still the same level of QoS as in the fully centralized model can be achieved.

Currently, SRP is a candidate for resource reservation in the hybrid configuration model but does not support all TSN features. RAP mitigates this issue, but does not yet address the hybrid configuration model in its current draft. A CNC for the hybrid model must provide a RAP-capable interface. As IEEE P802.1Qdj emerges, future CNCs will provide a RESTCONF/NETCONF based interface. Therefore, we propose an approach to implement the hybrid configuration model using an extended RAP, a novel RAP-CUC, and a RESTCONF based CNC in Section V.

IV. Overview of LRP and RAP for TSN Networks

We first give an overview of the Link-Local Registration Protocol (LRP) [14], as well as, its proxy models, and introduce the current state of RAP [11].

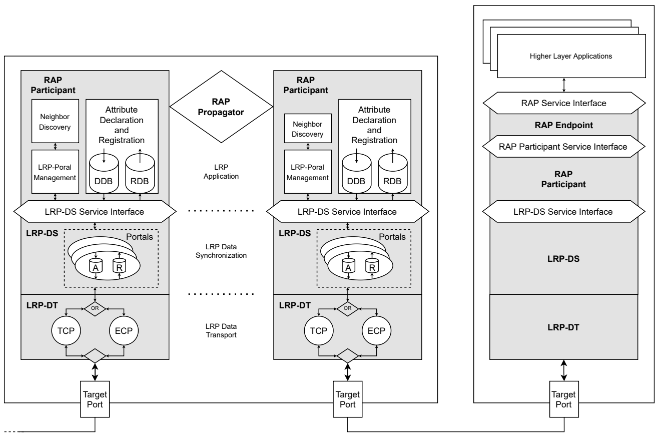


Figure 4: On the left side is a two port, RAP-capable bridge and on the right side is a RAP-capable end station.

A. Link-Local Registration Protocol (LRP)

LRP has been specified in IEEE P802.1CS [14] as transport protocol for TSN. We explain its architecture and signalling, system types and proxy models.

1) Architecture and Signalling: LRP is intended to transport LRP Data Units (LRPDU) hop-by-hop and to store this data persistently. LRP is similar to MRP but addresses scalability issues and adds new proxy mechanisms. LRP can be efficiently used to distribute databases of up to 1MB between communication peers, resolving scalability issues of MRP which was optimized for databases of up to 1500 Bytes [14].

Figure 4 illustrates a LRP bridge and end station with RAP as LRP application. The LRP protocol stack consists of three layers: LRP application, LRP Database Synchronization (LRP-DS), and LRP Database Transport (LRP-DT).

A LRP application implements application specific behavior. It can take forwarding decisions and configure bridge hardware based on received data. One or more LRP applications leverage the data synchronization service provided by LRP-DS.

LRP-DS establishes connections and controls data synchronization with LRP Portal instances. LRP Portals contain two databases for transmission and receipt of data: the Applicant Database and Registrar Database. The records of the Applicant Database of one system are copied to the Registrar Database of the peer. The connection managed by a Portal is bound to a single LRP application and to a physical port of a device, called target port. The target port is uniquely characterized by the chassis identifier and port identifier of a system. It is not required to reside on the same host as the Portal itself. For Portal creation, LRP-DS supports several methods, i.e., manual or protocol assisted configuration. Link Layer Discovery Protocol (LLDP) can be used for protocol assisted Portal creation. Therefore, the target port associated with a Portal advertises a list of applications and corresponding address information. Based on that, the two LRP devices establish a connection between their applications. As an alternative LRP introduces a special handshake for connection establishment, initiated by sending an Exploratory Hello LRP Data Unit (LRPDU).

LRP-DT transmits and receives LRPDU via Transmission Control Protocol (TCP) [15] or the Edge Control Protocol (ECP) [6]. ECP is a simple transport mechanism implementing flow control for the local link using a stop-and-wait automatic repeat request paradigm [17].

2) System Types: IEEE P802.1CS defines three system types.

i) A LRP system is native, when application, Portal and target port are physically located in the system. Native systems provide computational power for the applications, data storage capabilities for the Portals and a local target port.

ii) A proxy system implements an application and Portal instance. Additionally, it leverages the remote target port of a controlled system, e.g., for Portal creation via LLDP. The proxy systems can reside at a remote location like the edge of the network or in the cloud.

iii) A controlled system only provides a physical port as a remote target port for a proxy system, and does not have to implement LRP itself. A proxy system and a controlled system must be used in combination to form a functional LRP system.

Such a composite system only supports TCP for LRP-DT, and LLDP or manual configuration for Portal creation. The proxy system provides a list of application services and address information to the controlled system via network management. The controlled system advertises the provided information on its local target port via LLDP to its peer. With the advertised address and application information a peer-to-peer, TCP connection is established. This connection can be used to exchange application data between the peer's and the proxy system's application. The controlled system is not necessarily involved in data transport after the connection is established.

3) Proxy Models: IEEE P802.1CS proposes four proxy models using proxy/controlled systems for either relay systems, end systems, or both.

i) The "full native system" model, consists of a native bridge and a native end system. Connections can be established manually, with LLDP, and Exploratory Hello LRPDUs. LRP-DT can be based on ECP or TCP in that scenario. A use case for the "full native system model" is implementing the UNI for the fully distributed configuration model with RAP as LRP application.

ii) The "proxied relay systems" model is illustrated in Figure 5. It is composed of controlled relay systems providing target ports for a proxy system. This proxy model can be leveraged to implement the hybrid configuration model of TSN where the proxy system is part of the CNC.

iii) The "proxied end systems" model includes native relay systems, controlled end systems and an end systems' proxy. A use case can be incorporating legacy or simple end systems into a TSN network. The end systems' proxy handles, e.g., admission control on behalf of the end systems with the network.

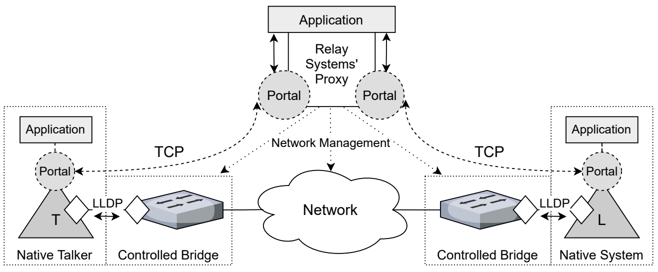


Figure 5: Hybrid configuration model implemented with LRP: the proxy system's application and address information is announced by the target port of the controlled bridges via LLDP. End stations discover the application of the proxy and establish a connection. The resulting TCP connection is used for data exchange between the LRP applications of the end stations and the proxy system's application.

iv) The "end systems' proxy and relay systems' proxy" model combines the two preceding proxy models. A use case for the model is the fully centralized configuration model of TSN. The end systems' proxy is the CUC and the relay systems' proxy is the CNC. The proxies exchange application data directly via TCP. The connection is set up by the controlled end stations and bridges physically connected. Both advertise address information of their proxy systems via LLDP to create Portals.

B. Resource Allocation Protocol (RAP)

RAP is a protocol for dynamic resource reservation for unicast and multicast streams. It is specified in IEEE P802.1Qdd Draft 0.4 [11] as a successor for SRP. RAP advances SRP to provide support for the recent evolution in TSN standardization, e.g., FRER. We illustrate a RAP-capable bridge and end station in Figure 4.

1) Domain Establishment with RAP: For resource reservation, all devices along the path of a stream must be members of the same RAP domain. A RAP domain comprises the set of neighbouring RAP devices which support a priority for a traffic class. This priority characterizes one of eight Resource Allocation classes (RA classes), along with a RA Class Template (RCT). The RCT describes a set of TSN mechanisms to be applied to streams of the class.

For domain establishment, each RAP capable device announces its RA classes link locally to its neighbours. Devices identify domain boundaries based on the priority values of the RA classes. The RCT is not evaluated for domain establishment, thus the use of different traffic shaping mechanisms along a path is possible.

2) RAP Attributes: For resource reservation RAP end stations exchange structured data, so called attributes. RAP defines three attributes, encoded as a Type-Length-Value (TLV).

i) As discussed in the previous section, all RAP end stations declare RA class attributes for domain establishment link locally.

ii) The Talker Announce Attribute (TAA) is sent by Talkers to its Listeners for conveying stream identification information and traffic specification. For traffic specification, RAP offers a TLV for token bucket based shaping, including minimum/maximum frame size, committed information rate, and committed burst size. An alternative is the MSRP traffic specification, including maximum number of frame size and maximum number of frames per interval. Additionally, the network uses the TAA to compute the wort-case latency of a path and give status information to the Listener.

iii) The Listener Attach Attribute (LAA) is declared by a Listener for communicating the interest in participating in a stream to the network. The network uses it to convey admission control status to the Talker. The status carried by LAAs can be: Listener Ready, Failed, or Partial Failed.

All attributes can be extended with organizationally specific TLVs for adding custom features to RAP.

3) Resource Reservation Process: A resource reservation process with RAP is initiated by a control application of an end station which requires QoS guarantees for its application data streams. As a result, Talkers declare TAAs and Listeners declare LAAs to request resources for streams from the network.

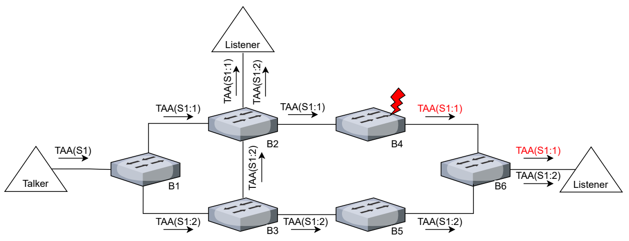
Figure 6(a) illustrates the signaling of Talker for reserving resources for a multicast stream via two disjoint paths. The Talker declares a TAA which is propagated by the bridges in direction of all Listeners along the path of the stream. On receipt of a TAA, bridges evaluate if sufficient resources for applying the requested QoS level are available. When resources are available, the attribute is forwarded in direction of the Listeners. In case of an error, failure information is attached to the TAA before forwarding. For computing worst-case path latency, bridges add their maximum forwarding latency to the accumulated latency field in the TAA.

Figure 6(b) illustrates the signaling of the Listeners. They declare a LAA to signal participation in a stream each. After a TAA and LAA of the same stream is registered on the ports of a bridge, resources for the stream are finally reserved and underlying QoS mechanisms are configured. When multiple LAAs for the same stream are received by a bridge, it merges the status information of all received LAAs before forwarding in direction to the Talker.

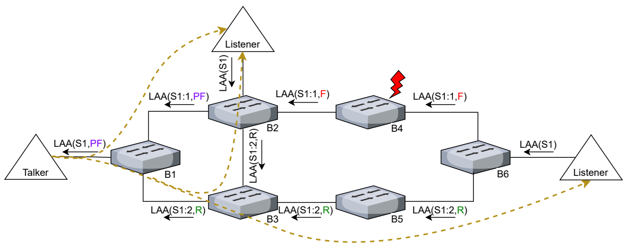
After a successful resource reservation, RAP notifies the control application which initiated resource reservation such that it can start deterministic data transmission.

V. Analysis And Extension of RAP for Hybrid Configuration Model

In this section we develop a concept for using RAP for admission control in the hybrid configuration model. First, we explain the problem statement for our scenario. We conclude that a RAP-CUC is needed and propose a general architecture for a CUC. We show the compatibility of RAP to the TSN UNI. Finally, we describe the design and implementation of a RAP-CUC.



(a) A Talker declares a TAA to signal stream requirements to the network. Based on that bridges pre-reserve resources. Bridge B4 is overbooked, therefore attaches failure information to the TAA to notify Listeners.



(b) Listeners declare a LAA to participate in a stream. Bridges merge status information of multiple LAAs for the same stream and forward one LAA in direction of the Talker. Additionally, resources are reserved and QoS functions are configured. The dashed arrow shows the path of the partially reserved realtime stream.

Figure 6: RAP signalling for reserving resources for an 1+1 protected, multicast stream.

A. Problem Statement

RAP standardization currently focuses on the fully distributed configuration model and mostly token bucket based traffic shaping. The hybrid configuration model requires a distributed signalling protocol, like RAP, and a CNC. Our goal is to enable resource reservation for globally scheduled realtime streams using RAP. We want to enable resource reservation for time-aware end stations in a TSN network following the hybrid configuration model. In our scenario the CNC provides a RESTCONF-based CUC-CNC interface.

In hybrid configuration model end stations' RAP requests are directly forwarded to the CNC by edge bridges. Since future CNCs will have a YANG based interface, we introduce an additional component, as a gateway between end stations and the CNC, to transform RAP-based requests to a format understandable by the CNC. Additionally, this component manages state of ongoing resource reservations. We state that such a component is equal to the CUC component known from fully centralized configuration model.

B. Life-Cycle of a TSN Stream

We assume that the life-cycle of a stream is managed by the CUC which keeps track of the state of streams and reacts to different events originating from CNC or end stations. We define the following states: *new*, *pending*, *deployed*, *withdrawn*, *error*.

The initial state of a stream is *new*, when either the requirements of a Talker or Listener is registered but not both.

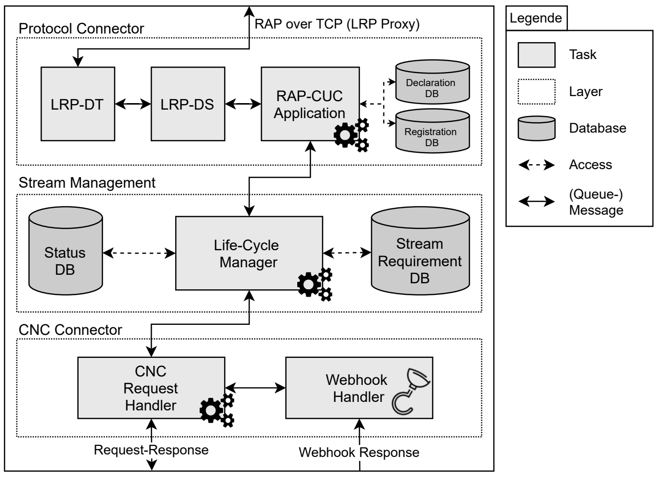


Figure 7: The proposed CUC consists of a three layer architecture: Protocol Connector, Stream Management, CNC Connector. The Protocol Connector is specific for RAP.

The stream reaches the *pending* state, when Listener and Talker requirements for a stream are received by the CUC. As a reaction, the CUC initiates a resource reservation process with the CNC.

When the CNC indicates a successful resource reservation, the state advances to *deployed*.

Otherwise, the *error* state is reached, e.g., due to insufficient resources. Additionally, the CNC can convey an error state to the CUC, even after the stream has been successfully deployed, e.g., on recognition of a link failure.

The state is *withdrawn* when either the Talker or all Listeners cancel their intent to participate in the stream. As a consequence, the CUC withdraws reserved resource from the CNC to free network resources.

C. Design of a General CUC Architecture

The CUC fulfills three tasks: communicate with end stations, manage the life-cycle of streams, and request resources for streams from the CNC. With regard to these tasks, we propose a three layer architecture consisting of a Protocol Connector, Stream Management, and CNC Connector. An example for RAP as user-specific protocol is depicted in Figure 7.

1) Protocol Connector Layer: The Protocol Connector communicates with end stations over a user-specific protocol. Thus, the implementation depends on the user-specific protocol and its signalling. The Protocol Connector extracts data relevant for admission control from user-specific protocol packets, transforms this data to group-talker or group-listener structures, and hands it to the Stream Management Layer (SML). In addition, it generates messages, as the user-specific protocol defines, to notify end stations about stream status changes and sends configuration data on behalf of the SML. We describe an implementation of a Protocol Connector based on RAP in Section V-E.

2) Stream Management Layer: The SML manages the whole life-cycle of a stream as described in Section V-B. Therefore, it stores stream requirements of end stations from Protocol Connector, manages stream reservation and withdrawal with the CNC Connector, and triggers notification of end stations. For that, it keeps track of end stations requirements in the Stream Requirement Database (SRDB) and stream reservation status in the Status Database (SDB). The SRDB stores the stream requirements per end station, either as IEEE Std 802.1Qcc group-talker or group-listener. The SDB contains a record for each stream currently under management. Each record contains information about the state of reservation process, a list of participating end stations, and the group-status-stream and group-status-talker-listener returned from CNC Connector on successful reservation.

SML is independent of the user-specific protocol and the CNC's implementation. Thus, the SML implements a general model for life-cycle management of streams in TSN.

3) CNC Connector Layer: The CNC Connector conveys stream requirements to the CNC and invokes remote procedure calls. It uses the groupings of IEEE Std 802.1Qcc, communicated by the SML, to initiate stream reservation/withdrawal process. For that, it communicates stream requirements, initiates computation and deployment of network configuration. The signalling performed by the CNC Connector is specific to the implementation of the CNC's interface.

Since schedule synthesis is a complex problem, it can take some time for larger networks [12]. A simple blocking, request-response paradigm or polling for the result is using resources inefficiently.

Therefore, we introduce the Webhook Handler for subscription-based result propagation. The Webhook Handler implements a REST-based application interface for providing callback addresses to the CNC. For each request, the CNC Connector obtains an Uniform Resource Identifier (URI) from the Webhook Handler. This URI will be sent along the request whose response is to be subscribed. The CNC transmits the computational result to the specified URI. Thus, a high number of open connections and blocking can be avoided.

D. Compatibility of RAP with TSN UNI

We analysed if RAP can be used for reserving resources for scheduled streams originating from time-aware Talkers. Therefore, we compared the minimum set of information, defined by the UNI, needed to take admission control decisions, with the attributes and signalling of RAP. Analysis has shown that RAP currently does not provide sufficient information to the CNC to take admission control decisions for the scheduled streams with time-aware end stations.

RAP lacks the possibility to communicate the interval length and the earliest/latest transmit time to the CNC. Additionally, choosing a custom latency bound of a stream for schedule synthesis is not possible using RAP. Furthermore, the resulting end station configuration, like the transmission start time of a Talker, can not be communicated to the end stations.

We propose to consider including the interval time, the earliest/latest transmit time, and the maximum latency as optional fields in the MSRP traffic specification. We also propose to allow attachment of interface configuration to the LAA and TAA attributes. These gaps can be immediately resolved by integrating the proposed extensions as organizationally defined TLVs in LAA/TAA.

E. Design of a RAP-specific Protocol Connector

We introduce a Protocol Connector which performs RAP specific signalling with the end stations. For connecting the end stations to the RAP-CUC, we leverage the "proxied relay systems" model of LRP as illustrated in Figure 5.

In that scenario the Protocol Connector for RAP is the LRP proxy system and the end stations are native LRP systems. The target port of the controlled edge bridge announces RAP as an application and corresponding address information to the end stations via LLDP. As an alternative, Portals can be manually configured. The advertised information is used by the applications to establish a TCP connection. This connection from the end stations to the Protocol Connector of the RAP-CUC enables exchange of RAP attributes for resource reservation.

The Protocol Connector includes an implementation of LRP-DT, LRP-DS and a newly introduced RAP-CUC application component. The RAP-CUC application extracts information relevant for admission control from the received TAA and LAA attributes. The information is passed to the SML for managing state of stream reservation. Additionally, withdrawal of attributes by end stations is conveyed, too. On behalf of SML, the RAP-CUC application updates the fields of the registered RAP attributes and controls forwarding for conveying admission control results to the end stations.

VI. Implementation of a RAP-specific Protocol Connector

We release an implementation of the described RAP-CUC [19]. The prototype follows the proposed general CUC architecture and consists of: the RAP specific Protocol Connector, SML, and a partial implementation of the CNC Connector.

The Protocol Connector uses manual configuration for LRP Portal creation. Since Portal creation results in a TCP connection from the end stations to the RAP-CUC, we omit implementation of LRP-DS and LRP-DT and directly establish the aforementioned TCP connection. This approach is sufficient for evaluating RAP as a resource reservation protocol for hybrid configuration model.

We successfully tested our implementation against a proprietary CNC which provides a RESTCONF-based interface. We can not disclose implementation detail about the interface of the CNC, therefore only a rudimentary CNC Connector is provided.

The published CNC Connector can be used as a basis for developing an other CNC Connectors for a specific CNC. Furthermore, the RAP specific Protocol Connector can be replaced by a custom Protocol Connector for connecting end stations via other user-specific protocols.

VII. Conclusion

We gave an introduction to LRP and RAP as a future resource reservation protocol of TSN. We found out RAP currently lacks the ability to reserve resources for time-aware scheduled streams in the hybrid configuration model. To resolve the issues, we proposed an extension to RAP and an approach for immediate remedy. We describe a general architecture for CUC components and a specific implementation of a RAP-CUC. The implementation of the RAP-CUC is published under an open-source license [19]. Future work comprises further investigation of use-cases for LRP proxy models.

References

[1] Home page - opc foundation. https://opcfoundation.org/. (Accessed on 04/14/2021).

[2] IEEE Standard for Local and Metropolitan Area Network–Virtual Bridged Local Area Networks – Amendment 12: Forwarding and Queueing Enhancements for Time-Sensitive Streams. *IEEE Std 802.1Qav-2009*, 2009.

[3] IEEE Standard for Local and Metropolitan Area Network–Audio Video Bridging (AVB) Systems. *IEEE Std 802.1BA-2011*, 2011.

[4] IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks – Amendment 25: Enhancements for Scheduled Traffic. *IEEE Std 802.1Qbv-2015*, 2015.

[5] IEEE Standard for Local and Metropolitan Area Network–Frame Replication and Elimination for Reliability. *IEEE Std 802.1CB-2017*, 2017.

[6] IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks. *IEEE Std 802.1Q-2018*, 2018.

[7] IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks – Amendment: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements. *IEEE Std 802.1Qcc-2018*, 2018.

[8] L. Lo Bello and W. Steiner. A Perspective on IEEE Time-Sensitive Networking for Industrial Communication and Automation Systems. *Proceedings of the IEEE*, 2019.

[9] A. Bierman, M. Bjorklund, and K. Watsen. RFC8040: RESTCONF Protocol, 2017.

[10] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. RFC1157: Simple Network Management Protocol (SNMP), 1990.

[11] Feng Chen. IEEE Draft Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks – Amendment: Resource Allocation Protocol. *IEEE Std P802.1Qdd Draft 0.4*, 2020.

[12] Frank Dürr and Naresh Ganesh Nayak. No-wait packet scheduling for IEEE time-sensitive networks (TSN). In *Proceedings of the 24th International Conference on Real-Time Networks and Systems - RTNS '16*. ACM Press, 2016.

[13] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. RFC6241: Network Configuration Protocol (NETCONF), 2017.

[14] Norman Finn. IEEE Draft Standard for Local and Metropolitan Area Network–Link-local Registration Protocol. *IEEE Std P802.1CS Draft 3.1*, 2020.

[15] Information Sciences Institute. RFC793: Transmission Control Protocol, September 1981.

[16] Stephan Kehrer. IEEE Draft Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks – Amendment: Configuration Enhancements for Time-Sensitive Networking. *IEEE Std P802.1Qdj*.

[17] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Pearson, 6 edition, 2012.

[18] J. L. Messenger. Time-Sensitive Networking: An Introduction. *IEEE Communications Standards Magazine*, 2018.

[19] Lukas Osswald. uni-tue-kn/rap-cuc. https://github.com/uni-tue-kn/rap-cuc. (Accessed on 05/05/2021).

2.5 Robust LFA Protection for Software-Defined Networks (RoLPS)

Robust LFA Protection for Software-Defined Networks (RoLPS)

Daniel Merling, Steffen Lindner, and Michael Menth
Chair of Communication Networks, University of Tuebingen, Germany
{daniel.merling, steffen.lindner, menth}@uni-tuebingen.de

***Abstract*—In software-defined networks, forwarding entries on switches are configured by a controller. In case of an unreachable next-hop, traffic is dropped until forwarding entries are updated, which takes significant time. Therefore, fast reroute (FRR) mechanisms are needed to forward affected traffic over alternate paths in the meantime. Loop-free alternates (LFAs) and remote LFAs (rLFAs) have been proposed for FRR in IP networks. However, they cannot protect traffic for all destinations and some LFAs may create loops under challenging conditions.**

This paper proposes robust LFA protection for software-defined networks (RoLPS). RoLPS augments the coverage of (r)LFAs with novel explicit LFAs (eLFAs). RoLPS ranks available LFAs according to protection quality and complexity for selection of the best available LFA. Furthermore, we introduce advanced loop detection (ALD) so that RoLPS stops loops caused by LFAs. We evaluate RoLPS-based protection variants on a large set of representative networks with unit and non-unit link costs. We study their protection coverage, additional forwarding entries, and path extensions for rerouted traffic, and compare them with MPLS facility backup. Results show that RoLPS can protect traffic against all single link or node failures, and against most double failures while inducing only little overhead. We implement FRR on the P4-programmable switch ASIC Tofino and provide a control plane logic based on RoLPS. Measurement results show that the prototype achieves a throughput of 100 Gb/s, reroutes traffic within less than a millisecond, and reliably detects and drops looping traffic.

***Index Terms*—Software-Defined Networking, P4, Loop-Free Alternates, Resilience, Link Protection, Node Protection, Scalability,**

I. Introduction

Software-defined networking (SDN) separates data plane and control plane of forwarding nodes. A controller computes and installs forwarding rules on data plane devices to instruct them how to process data packets. Packet forwarding is impaired when a next-hop becomes unreachable due to a failure, i.e., a failed link or a failed node. Without controller interaction, switches drop affected packets. However, notification of the controller, recomputation of forwarding rules, and their installation on data plane devices takes a considerable amount of time. This outage time is too long, in particular for the transport of realtime traffic.

In IP networks fast reroute (FRR) mechanisms are used to quickly reroute packets via pre-computed backup paths while forwarding entries are recomputed. FRR would also be helpful in SDN to forward traffic with unreachable next-hops without controller interaction via alternate paths. However, SDN forwarding devices often have limited forwarding tables so that adding many forwarding entries for FRR purposes may be problematic. Loop-free alternates (LFAs) are a well-known FRR method for IP networks that requires no additional forwarding entries so that we consider them in this work. LFAs constitute alternative next-hops that successfully forward traffic towards the destination when the default next-hop is unreachable. The authors of [1] proposed to use LFAs to protect traffic without controller interaction in SDN-based networks. However, LFAs suffer from two major shortcomings. First, they cannot protect traffic for all destinations against single link failures (SLF) and single node failures (SNF). Second, some LFAs may cause rerouting loops in case of node failures or multiple failures.

In previous work [2] we improved the usage of LFAs in software-defined networks. We introduced explicit LFAs (eLFAs) based on explicit tunnels to protect destinations that cannot be protected by other LFAs. We proposed advanced loop detection (ALD) to detect and stop loops, which prevents severe overload that may happen with LFAs in failure cases. We described loop avoidance (LA), which leverages ALD, ranks available LFAs according to their protection quality and overhead, and chooses the best one. Furthermore, we showed how LA can be implemented in OpenFlow. Finally, a simulation-based evaluation showed that LA can protect all traffic in SDN networks against SLF and SNF and with less overhead compared to other FRR methods.

his paper is an extension of [2] with the following advances. (1) We augment eLFAs with explicit multipoint-to-point rerouting tunnels. This significantly decreases the required number of additional forwarding entries for explicit tunnels. (2) We modify ALD so that it can detect and stop loops faster while being implementable on P4 devices. (3) We update the simulative evaluations according to the new mechanisms. (4) We include topology-independent LFAs (TI-LFAs) [3] in the simulative evaluations because they are conceptually similar to eLFAs. (5) We improved the overall presentation, including a renaming of LA into RoLPS as the name LA did not capture the entire concept. (6) We implement a prototype of RoLPS on the P4-programmable switching ASIC Tofino featuring LFAs, rLFA, eLFA, and ALD, and a RoLPS-based SDN controller, and thereby, show its technical feasibility. (7) We demonstrate that the technical solution performs well by showing that the prototype operates at 100 Gb/s, reroutes traffic within less than

The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/1-2. The authors alone are responsible for the content of the paper.

a millisecond, and reliably detects and drops looping traffic.

The paper is structured as follows. In Section II we discuss related work. Then, we review state of the art for LFAs in Section III. Section IV introduces eLFAs and ALD for improved protection of the SDN data plane, and a RoLPS-based control plane logic for that features and existing LFAs. Section V describes the simulative evaluation methodology and discusses performance results based on comprehensive study. We present the implementation of a P4-based hardware prototype in Section VI. We evaluate its performance in Section VII by measurements. Finally, we conclude the paper in Section VIII. A table of acronyms and a glossary are provided at the end of the paper to facilitate the reading.

II. Related Work

In this section we describe related work. First, we discuss legacy FRR mechanisms to position LFAs. Then, we review FRR for SDN.

A. FRR in Legacy Networks

Rai et al. [4], Raj et al. [5], and Papan et al. [6] present surveys that provide a wide overview of FRR in legacy networks. Hutchinson et al. [7] discuss the architecture and design of resilient network systems, i.e., specifying and realizing appropriate components. They review state-of-the-art contributions and identify future research issues.

1) MPLS Networks: For MPLS [8] two major FRR mechanisms have been proposed [9]. One-to-one backup reroutes packets on preconfigured paths that avoid the failure. Facility backup tunnels the packets locally around the failure to the next-hop for link protection, or to the next-next-hop for node protection. Only recently, the authors of [10] propose a loop detection mechanism for MPLS. It is based on special MPLS labels that are pushed on the MPLS header stack when a packet is rerouted. This allows nodes to detect whether a packet has already been rerouted.

2) IP Networks: Not-via addresses [11] protect both IP and MPLS networks. The routing table of a node contains one additional forwarding entry for every outgoing link. When the default next-hop is unreachable, those additional entries are used to deviate the packet from its shortest path through a tunnel around the failure. This causes a similar path layout as MPLS facility backup [12]. Failure insensitive routing (FIR) [13] leverages interface-specific routing tables to encode failure information. Depending on the ingress interface, packets are rerouted on precomputed backup paths around the failure. Multiple routing configurations (MRCs) [14] implement multiple disjoint routing topologies so that always at least one topology provides a working path towards the destination despite the failure. For each topology, an entire set of forwarding entries is required which at least doubles the amount of forwarding entries. Maximally redundant trees (MRTs) [15] leverage a similar approach. A red and a blue set of backup forwarding entries are computed so that at least one set delivers the packet in case of a failure. However, MRTs triple the number of forwarding entries in the network and may lead to extensive backup paths [16]. LFAs can be combined with MRTs to reduce backup path length and link load [17]. Independent directed acyclic graphs (IDAGs) [18] compute only two sets of maximally disjoint forwarding entries, i.e., doubling the amount of forwarding entries so that one is working in case of a failure. The authors of [19] encode failure information in the packet header. Nodes leverage this information to identify the failure and reroute packets on disjoint paths around it.

3) LFA-Based Protection: LFAs [20] with either link or node protection locally reroute packets around the failure on shortest paths. Therefore, they do not require additional forwarding entries but cannot protect all destinations. Csikor et al. [21], [22] increase the number of protected destinations by optimizing link costs. rLFAs [23]–[25] augment LFAs to increase the number of protected destinations by rerouting packets to remote nodes through shortest path tunnels. They do not need additional forwarding entries but still cannot protect all destinations. The performance of both LFAs and rLFAs can be enhanced by adding links to the network [26]. In [27], the authors present a self-configuring extension for LFAs based on probes. It installs alternative hops in other nodes to prevent rerouting loops. Topology-independent LFAs (TI-LFAs) [3] leverage segment routing (SR) [28] to protect against failures. SR is based on forwarding instructions in the packet header which may be stacked. TI-LFAs leverage SR to implement explicit tunnels to remote nodes. As eLFAs leverage explicit tunnels, too, they can be viewed as a very specific but rather untypical form of TI-LFAs.

B. FRR Protection in SDN

We discuss FRR in the context of SDN. We first address general FRR approaches for SDN and then we discuss related work for FRR in OpenFlow- and P4-based networks.

1) FRR in SDN: There have been many proposals to make the SDN control plane more resilient [29]. However, there are only very few efforts to protect traffic in the data plane. If the controller is notified about the failure, it may update its topology, and recompute and install updated forwarding entries. Sharma et al. [30] measure that recomputation takes about 80-100 ms. However, the authors clarify that this number highly depends on the number of affected flows, path lengths, and traffic bursts in the control network. In particular, it is likely that the time for rerouting is significantly higher in larger networks. Da Silva et al. [31] and Chiesa et al. [32] present surveys that give overviews of FRR in SDN with significantly faster protection than recomputation of forwarding entries.

2) OpenFlow-Based FRR: FRR capabilities have been introduced in OpenFlow with Version 1.1. The authors of [33] provide a BFD-based protection scheme for earlier OpenFlow versions than 1.1. It is based on a bidirectional forwarding detection (BFD) where nodes periodically exchange information about their reachability. Van Adrichem et al. [34] measure that failure detection takes about 3-30 ms on the software-based Open vSwitch depending on the configuration of the BFD. SlickFlow [35] encodes primary and backup paths in the packet header to reroute packets when an unavailable egress port is selected. SPIDER [36] leverages additional state in the OpenFlow pipeline. Packet labels carry reroute and connectivity information. Braun et al. [1] propose loop detection for LFAs (LD-LFA) which increases the number of protected destinations but may erroneously drop packets. The authors of [37] use labels in the packet header that carry failure information to trigger rerouting in other nodes. Cevher et al. [38] implement MRCs in OpenFlow. The authors of [39] implement multi-topology routing which uses virtual topologies to provide redundancies in routing tables. If a failure is detected, packet forwarding is switched to a topology which is not affected by the failure. BOND [40] optimizes memory management for backup rules and leverages global hash tables to accelerate failure recovery.

3) P4-based FRR: P4 does not provide native FRR capabilities. Therefore, the hardest challenge is to provide the data plane devices with information about which neighbors are reachable, i.e., which port is up or down.

Sedar et al. [41] propose to use registers to store information about which egress port is up or down. Depending on the port status registers, primary or backup forwarding actions are triggered. However, the authors depend on a local agent to populate the registers. Shared Queue Ring (SQR) [42] caches recent traffic in a delayed queue. If a link failure is detected, the cached traffic is sent over alternative paths. Lindner et al. [43] implement 1+1 protection in P4 which replicates traffic, includes sequence numbers, and sends it over disjoint paths. The joint head end of those paths deduplicates the traffic. Hirata et al. [44] implement a FRR scheme in P4 which is similar to MRCs. Multiple routing topologies with disjoint paths are deployed. A field in the packet header identifies the topology which should be used for forwarding. D2R [45] is a resilience mechanism which works entirely in the data plane. When a failure is detected, the data plane itself, i.e., the failure-detecting switch, recomputes a new path to the destination. A primitive for reconfigurable fast reroute (PURR) [46] stores additional egress ports for each destination. During packet processing, the first working egress port is selected for forwarding.

III. LFAs: State of the Art

We review LFAs and remote LFAs (rLFAs) and give an overview of previous work regarding loop detection for LFAs. Finally, we explain topology-independent LFAs (TI-LFAs).

A. LFAs and rLFAs

In this subsection we introduce the concept of LFAs and rLFAs. Then, we discuss three important properties of LFAs. First, we differentiate protection levels for LFAs, i.e., link protection and node protection. Second, we explain the influence of links cost on LFA-based protection. Third, we point out that LFAs may generate loops under some conditions.

1) Concept: LFAs [20] have been proposed in the context of FRR for IP networks to quickly protect traffic against the failure of links and nodes while primary forwarding entries are recomputed.

A point of local repair (PLR) denotes a node that detects an unreachable next-hop and reroutes affected traffic to some other neighbor. However, some neighbors would send the traffic back to the PLR, which creates a loop. The other neighbors can forward the traffic without creating a loop and are called loop-free alternates (LFAs). They are used by a PLR to reroute traffic in case of a failure.

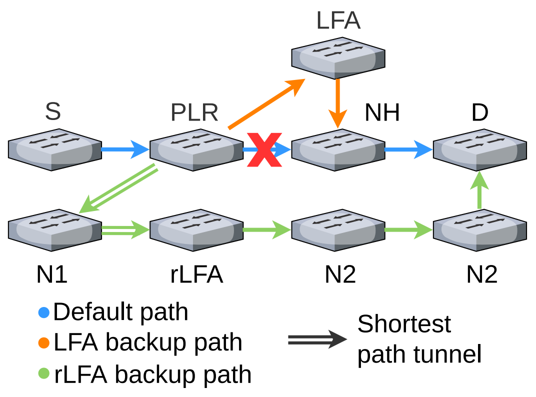


Figure 1: In case of a failure, a PLR may reroute a packet to an LFA or tunnel it via a shortest path to a rLFA. The (r)LFA then forwards the packet via a shortest path to its destination.

LFAs are illustrated in Figure 1. Traffic is forwarded on shortest paths. A packet is sent from sender *S* to destination *D*. The default path is via *PLR* and *NH*. When *PLR* cannot reach its next-hop *NH* due to a link failure, it cannot reroute the packet via neighbors *S* or *N1* as they forward traffic towards *D* to *PLR*, which creates a loop. However, *PLR* may reroute the packet via *LFA* which can forward the packet to *D*. Thus, the node *LFA* represents an LFA for *PLR* with respect to destination *D*.

We now assume that *NH* fails so that *LFA* has no working path towards *D*. If *PLR* reroutes the packet to *LFA*, *LFA* may use *PLR* as an LFA and return the packet. Thus, a loop occurs.

Remote LFAs (rLFAs) [23]–[25] have been introduced to protect more destinations than LFAs by sending packets through shortest path tunnels to remote nodes. In our example, the node *rLFA* is an rLFA for *PLR* with respect to destination *D*. If *NH* fails, *PLR* may tunnel the packet to *rLFA* which decapsulates the packet and sends it to *D* via a shortest path.

2) Protection Level: We already observed that some (r)LFAs protect only against link failures, others protect also against node failures. The first are classified as link-protecting (LP), the second as node protecting (NP). A link-protecting LFA (LP-LFA) forwards traffic to a destination via a path that avoids a PLR's failed link. A node-protecting LFA (NP-LFA) forwards traffic to a destination via a path that avoids a PLR's failed next-hop. Thus, NP-LFAs are also LP-LFAs, but not vice-versa. Therefore, a PLR can protect more destinations with LP-LFAs than with NP-LFAs. For some destinations, there may be no LP-LFA or NP-LFA at all. Then, rLFAs may help.

3) Influence of Link Cost: Networks are configured without link costs, i.e., unit link cost networks, or with link costs, i.e., non-unit link cost networks, e.g., for traffic-engineering. (r)LFAs have different protection properties in unit link cost networks than in non-unit link cost networks. The authors of

[1], [2] showed that for some destinations there is no LP-LFA or NP-LFA in both unit-link cost networks and non-unit link cost networks. Then, some destinations may be protected with rLFAs. Csikor et al. [25] proved that there is always an LP-rLFA for any destination in unit link cost networks. However, in [2] we showed that this is not the case in non-unit link cost networks. Furthermore, we showed that in both unit link cost networks and non-unit link cost networks there is not always an NP-rLFA for a destination [2] although there are more NP-rLFAs in unit link cost networks. Thus, in general, more destinations can be protected in unit link cost networks with (r)LFAs than in non-unit link cost networks.

4) LFA-Generated Loops: Forwarding loops in networks are problematic for two reasons. First, the traffic cannot reach its destination. Second, looping traffic consumes bandwidth, which may lead to packet loss for other traffic. However, looping traffic does not loop forever because the TTL field in the IP header limits the number of forwarding hops. As TTL=64 is a typical value, looping traffic can easily waste the 30-fold of the capacity it would normally occupy on a link. Therefore, routing loops are detrimental and should be avoided.

Depending on their protection level (r)LFAs may cause rerouting loops in specific failure scenarios. We distinguish and order four failure scenarios: single link failure (SLF) $<$ single node failure (SNF) $<$ double link failure (DLF) $<$ single link and single node failure (SLF+SNF).

LP-(r)LFAs do not cause rerouting loops for SLF but they may cause loops in other scenarios. NP-(r)LFAs prevent loops for both SLF and SNF [2], but fewer destinations can be protected by them. In case of multiple failures, even NP-(r)LFAs may generate loops. Some LP- or NP-(r)LFAs have the “downstream” property [12] and they avoid loops in case of multiple failures. However, only a few LFAs have that property so that only a few destinations can be protected by them. We do not consider them any further in this study.

B. Loop Detection for LFAs

The authors of [1] propose loop detection based on bit strings. They use it in combination with LFAs to protect more destinations by LFAs without suffering from loops. In addition, they suggest to protect destinations with LFAs with the highest possible protection level to maximize the coverage against link and node failures. They call this approach LD-LFA.

1) Loop Detection Based on Bit Strings: The loop detection in [1] requires a bit string in the packet header to indicate nodes that have rerouted the packet before. Each node in the network is associated with a bit position. If a packet is rerouted, the node activates it bit in the packet's header. If a node receives a packet with its corresponding bit activated, the packet is dropped.

The authors suggest an implementation in OpenFlow but do not deliver a prototype. An advantage of this approach is that a packet can be rerouted by multiple nodes. A disadvantage is the missing scalability. Bit strings in packet headers should be small. In OpenFlow, MPLS labels may be reused for that purpose, but they are only 4 bytes long which is not enough to number all nodes of a large network. Therefore, multiple nodes may be associated with the same bit. If one of these nodes reroutes a packet, the packet is dropped if it is received by another of those nodes. This causes erroneous drops for rerouted packets.

2) LFA Selection: For some PLRs there are several LFAs available for a specific destination. The authors of [1] suggested to prefer NP-LFAs over LP-LFAs in such a case. They showed for various network topologies that significantly fewer destinations can be protected by NP-LFAs than by LP-LFAs. Therefore, they suggested to protect the remaining destinations with LP-LFAs if possible. In addition, they proposed to utilize loop detection based on bit strings to avoid rerouting loops caused by LP-LFAs. They did not consider rLFAs.

C. Topology-Independent LFAs

In this subsection we explain topology-independent LFAs (TI-LFAs) [3]. First, we review segment routing (SR) [28]. Then, we describe TI-LFAs.

1) Segment Routing: IP networks leverage destination-based forwarding to deliver packets. That is, a packet carries the IP address of the destination in its header which is used by network devices to determine the appropriate next-hop according to entries in a forwarding table. In contrast, with SR the packet source determines the processing of a packet. To that end, SR leverages forwarding instructions in the packet header. The packet source constructs a set of header segments that are added to the packet. Each header segment corresponds to a specific action. Nodes process a packet according to the segments in its header. To that end, network devices maintain a certain number of forwarding entries to map a header segment to a specific action.

Currently, there are two major technologies that implement SR. SRv6 [47] is based on IPv6 and its extension header. Each IPv6 address in the extension header corresponds to one header segment. SR-MPLS ([48]) leverages stacked MPLS labels, i.e., the header stack, where each MPLS label is a header segment. To facilitate readability we only use the terminology of SR-MPLS, i.e., header stack and label, in the following.

Header segments may instruct nodes to perform arbitrary actions, e.g., forwarding a packet, pushing or removing other header segments, etc. In the following we focus on two specific types of header segments. The first type are header segments for global forwarding. We refer to such header segments with the term “global labels”. Global labels instruct the nodes to forward a packet according to shortest paths towards a specific destination. As a result, a global label is similar to destination-based forwarding in IP networks. At the destination the global label is removed and the node processes the next header segment. When global labels are used for all destinations, every nodes requires $n-1$ forwarding entries where n is the number of destinations in the network. The second type are header segments for local forwarding. We refer to that kind of header segments with the term “local labels”. Local labels instruct nodes to forward a packet over a specific link towards a next-hop. Before a node forwards a packet to the NH, it removes its local label from the header stack. When local labels are used for all nodes, every node requires d forwarding entries where d is the number of neighbors of that node.

A source may construct a header stack that contains both global labels and local labels. As a result, forwarding differs depending on which type of label is on top of the header stack. On some subpaths the packet is forwarded according to a global label and on some subpaths the packet is forwarded according to a local label.

2) Concept of TI-LFAs: TI-LFAs leverage SR to forward packets on explicit paths around a failure. That is, TI-LFAs are not restricted to shortest paths because they construct a header stack with explicit forwarding instructions so that the packet avoids the failure. As a result, TI-LFAs with LP protect against any single link failure independently of link costs, and TI-LFAs with NP protect against any single node failure independently of link costs. However, multiple header segments may be necessary which increases the size of the header stack and thereby the overhead in terms of additional packet headers. The authors of TI-LFAs state that "in an MPLS world, this may create a long stack of labels to be pushed that some hardware may not be able to push." ([3], 2021, p. 6).

The size of a specific header stack depends on how the explicit backup path is implemented. The straightforward approach is to use one explicit forwarding instruction for every hop, i.e., local labels. However, this requires one header segment for each hop which causes large header stacks. The size of the header stack can be reduced if subpaths of the explicit path are implemented with already existing global labels. That is, one global label replaces multiple local labels. This is possible when working shortest paths are subpaths of the explicit path. However, this may not be possible for all subpaths because sometimes no working shortest subpath is available due to the failure.

The authors of [3] do not specify how the header stack to implement explicit paths is built. In particular, this is an optimization that highly depends on the failure scenario, topology, link costs, and path selection. Therefore, we see research potential for the optimization of the TI-LFA header stack. This, however, is out of scope of this document. In the following we assume that TI-LFAs implement explicit paths only with local labels.

IV. Robust LFA Protection for Software-Defined Networks (RoLPS)

LFAs originated from IP networks. They are attractive for SDN because they entail only little overhead in terms of additional forwarding state. However, they have three major shortcomings. They have been designed only for shortest-path routing based on link costs, they cannot protect all destinations, and they may cause loops under some conditions.

In the following we explain how LFAs can be applied in SDN which allows for general destination-based forwarding. We present explicit LFAs so that all destinations can be protected in case of a failure, provided they can be physically reached by a working path. We describe an advanced loop detection method to detect and stop loops and prevent erroneous packet drop after up to n reroute actions. Finally, we propose how to utilize these components and consider different protection variants.

A. Applicability of LFAs for SDN

In the context of IP networks, equations considering link costs are used to classify neighboring nodes into non-LFAs, LP-LFAs, and NP-LFAs with regard to some destination [12]. Forwarding in SDN does not need to follow shortest path routing based on link costs, but general destination-based forwarding may be applied. Therefore, we briefly explain how (r)LFAs can be used in that context. Essentially, we need to classify neighboring nodes into no-LFAs, LP-LFAs, and NP-LFAs. A PLR's neighboring node is

- no LFA if its standard forwarding procedure forwards the traffic to the destination via a path containing the PLR.
- an LP-LFA if its standard forwarding procedure forwards the traffic to the destination via a path that does not contain the link from PLR to its next-hop towards the destination.
- an NP-LFAs if its standard forwarding behavior forwards the traffic to the destination via a path that does not contain the PLR's next-hop towards the destination.

This definition can be applied to normal LFAs, rLFAs, and to eLFAs that are presented later in this section.

Path computation is not a focus of this paper. To limit the parameter space for ease of understanding, we consider in the evaluation in Section V link-cost-based forwarding which is a special case of the more general destination-based forwarding.

B. Explicit LFAs

We first give an example where (r)LFAs cannot protect a destination. Such destinations can be protected by explicit LFAs (eLFAs) which are based on explicit tunnels. However, explicit tunnels require additional forwarding entries. In [2] we suggested to implement explicit tunnels with explicit point-to-point rerouting tunnels. In this paper, we propose explicit multipoint-to-point rerouting tunnels as an alternative which requires significantly less additional forwarding entries. Finally, we explain the relation between eLFAs and TI-LFAs.

1) Protection through Explicit Tunnels: The network in Figure 2 forwards traffic on shortest paths based on costs that are annotated on the links. *PLR* sends a packet to *D* but the primary next-hop is unreachable. Although there is a physical path via *N1* and *eLFA*, there is no (r)LFA available. *N1* is not an LFA because it sends traffic to *D* via *PLR*. *eLFA* cannot serve as rLFA because the shortest path from *PLR* to *eLFA* traverses *D*. The problem can be solved by setting up an explicit tunnel via *N1* to *eLFA* a priori. If *D* is no longer reachable, *PLR* can send the packet over that explicit tunnel, and from *eLFA* the packet reaches *D* via a shortest path. Thus, *eLFA* is an eLFA for *PLR* with regard to *D*.

2) Explicit Point-to-Point Rerouting Tunnels: Now we explain the concept of explicit point-to-point tunnels which we introduced in [2]. In Subsection VI-C3 we describe technical details about the implementation of explicit tunnels in general with P4.

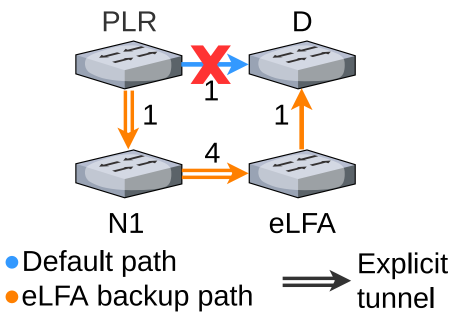


Figure 2: In case of a failure, a PLR may reroute a packet to an eLFA via an explicit tunnel which then forwards the packet via a shortest path to its destination. In contrast to rLFAs, the PLR cannot reach the eLFA via a shortest path.

Explicit point-to-point rerouting tunnels do not follow standard paths. Therefore, they are configured with a unique identifier, e.g., a unique number or IP address, in advance. When a PLR reroutes a packet through an explicit point-to-point rerouting tunnel, it adds the identifier of that tunnel to the packet. Nodes use the identifier to forward the packet along the explicit path. To that end, the nodes along an explicit path need additional forwarding entries for the identifier of that tunnel. Additional forwarding entries for FRR purposes are undesired overhead for the data plane as they limit its scalability.

3) Explicit Multipoint-to-Point Rerouting Tunnels: The overhead of additional forwarding entries from explicit tunnels can be reduced by using explicit multipoint-to-point tunnels. That is, the explicit tunnels from multiple PLRs towards the same endpoint, i.e., an eLFA, build a destination tree where the PLRs are the sources and the eLFA is the sink. Such an explicit multipoint-to-point rerouting tunnel corresponds to a specific eLFA and is identified by a single unique identifier. When a PLR reroutes a packet towards a specific eLFA, it adds the identifier of the corresponding multipoint-to-point rerouting tunnel to the packet. As a result, overlapping subpaths of explicit tunnels towards the same eLFA require only a single additional forwarding entry in nodes along that subpath. Therefore, multipoint-to-point rerouting tunnels are prefered over point-to-point rerouting tunnels. We evaluate the effect of multipoint-to-point rerouting tunnels in comparison to point-to-point rerouting tunnels in Section V-C.

4) Relation to TI-LFAs: Explicit tunnels can be implemented in different ways. We suggest eLFAs which implement explicit tunnels with a single tunnel header and additional forwarding entries in forwarding devices. Alternatively, TI-LFAs leverage a header stack with explicit forwarding instructions based on already available forwarding entries. Section III-C contains details about the construction of the TI-LFA header stack. Either way creates overhead to implement explicit tunnels. In Section V-C we evaluate the number of additional forwarding entries that are required by eLFAs. In Section V-D we quantify the size of the packet header stack when TI-LFAs are used.

C. Advanced Loop Detection

The loop detection method in [1] suffered from scalability problems. Therefore, we propose that packets are dropped if they are rerouted more than n times. This requires only a counter in the packet header which is increased with each reroute action. When the counter reaches the limit, the packet is dropped. We denote this advanced loop detection (ALD). Generally, ALD can be configured to support an arbitrary number of redirects. However, a large number can be counter-productive as packets are dropped later in case of loops and consume more bandwidth. In our context, we allow a packet to be rerouted twice so that double failures can be survived.

1) Implementation in OpenFlow: Due to technical restrictions of OpenFlow, conditions can be checked only at the beginning of the forwarding pipeline. However, at that stage, there is no knowledge about the packet's next hop and failed interfaces. Fortunately, it is possible to increase the reroute counter while rerouting. Thus, only the next-hop of a rerouted packet can determine whether the packet's reroute counter exceeds the limit and then the packet is dropped. This wastes bandwidth on the last link over which the packet was rerouted.

We provided a more detailed sketch of an OpenFlow-based implementation of ALD in [2]. That particular proposal was still based on bit strings. However, it avoids erroneous packet drops after a single reroute in contrast to the solution in [1].

2) Implementation in P4: P4 offers more implementation flexibility. Therefore, it is possible to check whether a packet is rerouted and whether its rerouting counter exceeds the limit before the packet is forwarded to the egress port. As a consequence, packets are dropped before transmission, which does not waste bandwidth. More details about the P4-based implementation of ALD are given in Section VI-D.

D. RoLPS Protection Variants

With SDN a controller configures flow entries on data plane devices. Alternative paths can be configured so that the device can switch over to a secondary next-hop if the first hop becomes unreachable. The secondary next-hop is also configured by the controller. In this section we present a ranking scheme for LFAs to choose the best one as a secondary next-hop. We further define protection variants and propose a corresponding nomenclature.

1) LFA Ranking: A controller can classify neighboring and remote nodes of a potential PLR into LFAs, rLFAs, and eLFAs, and as LP or NP for a specific destination. These LFAs can be ranked according to their protection level, i.e., NP is better than LP. Recall that NP-LFAs are also LP-LFAs, but not

Rank	LFA Type
0	NP-LFA
1	NP-rLFA
2	NP-eLFA
3	LP-LFA
4	LP-rLFA
5	LP-eLFA

Table 1: Ranking of LFA types according to protection level and complexity. Preference is given to LFAs with lower rank number.

Mechanism	C-LFA (nLD-LP-LFA)	C-rLFA (nLD-LP-rLFA)	LD-LFA (ALD-NP-LFA)	ALD-NP-rLFA	ALD-LP-eLFA	ALD-NP-eLFA
Loop detection			●	●	●	●
Protection against all SLF		o		o	●	●
Protection against all SNF						●
Additional forwarding entries					●	●

Table 2: Properties of protection variants.
Legend: o = only for unit link costs; ● = independent of link costs.

vice-versa. They can also be ranked according to complexity. Normal LFAs are simplest as they do not require tunneling. eLFAs are most complex as they entail additional forwarding entries for explicit tunnels.

With SDN, it is important to have an alternative next-hop in case the primary next-hop is unreachable as it may take too long until the forwarding is fixed by the controller. Therefore, we rank LFAs first according to their protection level and then according to their complexity. This yields the ranking given in Table 1. The ranking is used to select the best available LFA during computation.

2) Protection Variants: We define several protection variants with respect to loop detection, LFA complexity, and protection level. The following naming scheme is used: {nLD, ALD}-{LP, NP}-{LFA, rLFA, eLFA}. Loop detection may be activated or not {ALD, nLD}. Either the LP property is sufficient or NP is desired {LP, NP}. Only normal LFAs may be allowed, normal and rLFAs may be allowed, or normal, remote, and explicit LFAs are supported {LFA, rLFA, eLFA}.

eLFAs are preferably implemented with explicit multipoint-to-point rerouting tunnels (see Section IV-B3). However, for comparison we sometimes refer to eLFAs with point-to-point tunnels. To that end, we add the suffix "-p2p" to the protection variant. We omit a suffix for eLFAs with multipoint-to-point rerouting tunnels because this is the preferable way That is, *-*-eLFA refers to protection variant with eLFAs with multipoint-to-point rerouting tunnels and *-*-eLFA-p2p refers to protection variants with eLFAs with point-to-point rerouting tunnels.

If a protection variant requires the NP property, the LFA selection process starts with the search for an LFA of rank 0. If the search is successful, this LFA is configured as secondary next-hop for a specific destination, and the algorithm stops. Otherwise the search continues with the next higher rank number. This possibly continues up to rank 5. That means, NP-(e/r)LFAs are preferentially utilized, but LP-(e/r)LFAs may be used if the destination cannot be protected otherwise. This is needed, e.g., if the protected next-hop is the destination. If no LFA has been found for the last rank, there is no physical connection between PLR and destination.

If a protection variant requires only the LP property, the LFA selection process starts with the search for an LFA of rank 3. The algorithm also stops if no LFAs has been found for the last rank. In that case there is no physical path between PLR and destination. Note that LFAs of rank 3 may also be NP as every NP-LFA also fulfills the LP property. LP-LFAs are just not preferred over NP-LFAs when the protection variant requires only the LP property.

Protection variants requiring the NP property may still suffer from loops since some destinations can be protected only with LP-(e/r)LFAs. For example they occur when the destination of a flow fails. nLD-LP-LFA and nLD-LP-rLFA leverage only the classic LP-LFAs [20] and LP-rLFAs [23]. They are widely used in IP networks and we denote them as the classic LFA and rLFA variants (C-LFA, C-rLFA). ALD-NP-LFA[1] has been investigated as a preferred protection variant in [1] under the name LD-LFA.

Table 2 summarizes the most important protection variants investigated in our study. It summarizes properties regarding protection level and complexity. ALD-mechanisms prevent loops in any failure scenario. *-*-rLFA protect against all protectable SLF in networks with unit link costs. *-*-eLFA methods achieve that protection level even in networks with non-unit link costs. *-NP-eLFA protects even against all protectable SNF in networks with either unit or non-unit link costs.

V. Simulative Performance Evaluation of LFA-Based Protection

In this section we analyze the efficiency of LFA-based FRR mechanisms. First, we describe the methodology. The performance metrics of interest are protection coverage, required amount of additional forwarding entries, required amount of header segments for TI-LFAs, and path lengths. We compare them for RoLPS protection variants and other well-known FRR mechanisms. Finally, we discuss the presented results.

A. Methodology

We explain the methodology for the simulation-based evaluation. We describe the general approach, and discuss the topology data set and link costs used in the evaluation.

1) General Approach: We take a network topology including link costs and a RoLPS protection variant as input parameters. Then we compute LFAs according to Section IV-D. We evaluate different protection variants against various sets of failure scenarios, i.e., $\mathcal{S} \in \{\mathrm{SLF}, \mathrm{SNF}, \mathrm{DLF}, \mathrm{SLF{+}SNF}\}$ (see Section III-A4). To that end, we consider all source-destination pairs $f \in \mathcal{F}$ in the network and analyze how their traffic is forwarded in a specific failure scenario $s \in \mathcal{S}$.

Although RoLPS works for general destination-based forwarding (see Section IV-A), we limit the evaluation to shortest paths routing based on link costs to reduce the parameter space.

[1]Approximation of LD-LFAs with better loop detection.

2) Network Topologies: We evaluate 205 wide area, commercial, research, and academic networks from the Internet topology zoo [49] and three typical data center topologies (fat-tree, DCell, BCube) which were studied in [1]. For each topology we calculate both average values and maximum values for the considered metrics. We explain these metrics in Sections V-B1, V-C1, and V-E1. We visualize the results in bar diagrams or complementary cumulative distribution functions (CCDFs).

3) Link Costs: In Subsection III-A3 we showed that link costs have a significant impact on the protection properties of LFAs. To account for that fact, we perform evaluations on then networks with both unit link cost and non-unit link cost. However, the topology zoo does not include link costs for all networks. Therefore, we calculate link costs on all networks as proposed in [50]. For each link we derive the specific load based on a homogeneous traffic matrix, shortest paths, and unit link costs. The link cost of each link is the inverse of its load multiplied by the largest link load in the network so that the smallest link cost is 1. Over all topologies this leads to an average link cost of 6.8 and a coefficient of variation of link costs of 1. Thus, the generated link costs differ substantially.

B. Protection Coverage

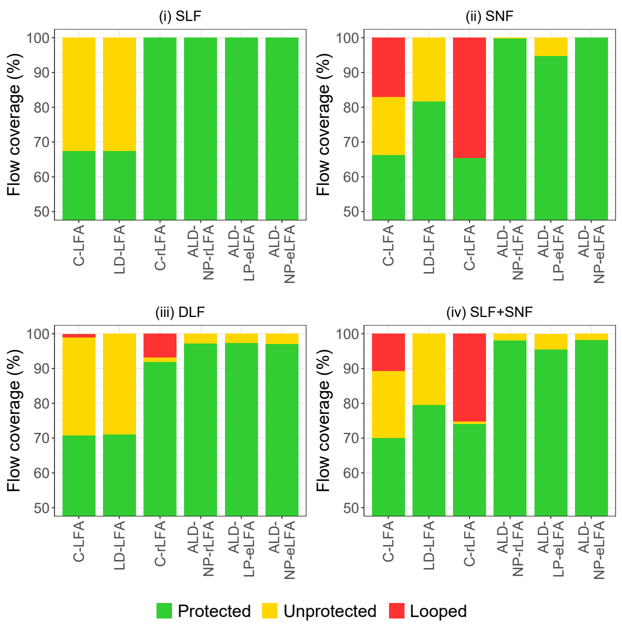
In this subsection we evaluate and compare the coverage of RoLPS protection variants. First, we explain the metric. Then, we briefly describe the evaluated protection mechanisms. Finally, we discuss results for networks with unit link costs and with non-unit link costs.

1) Metric: We introduce the three terms 'protected', 'unprotected', and 'looped' to refer to the quality of protection which is provided by a FRR mechanism for a flow in a specific scenario that consists of topology, failure scenario, and link costs. A flow is considered protected in two cases. First, if the packet is still successfully delivered at the destination although the path from source to destination was interrupted by a failure. Second, if a packet is dropped to prevent a loop because the destination is not reachable anymore. A flow is unprotected if the packet is dropped although the destination is still reachable. Finally, a flow is denoted as looped if a microloop was caused by local rerouting. We report the average fraction of protected, unprotected, and looped flows over all 208 topologies (see Section V-A2) in bar diagrams. The term coverage refers to the fraction of protected flows in a scenario.

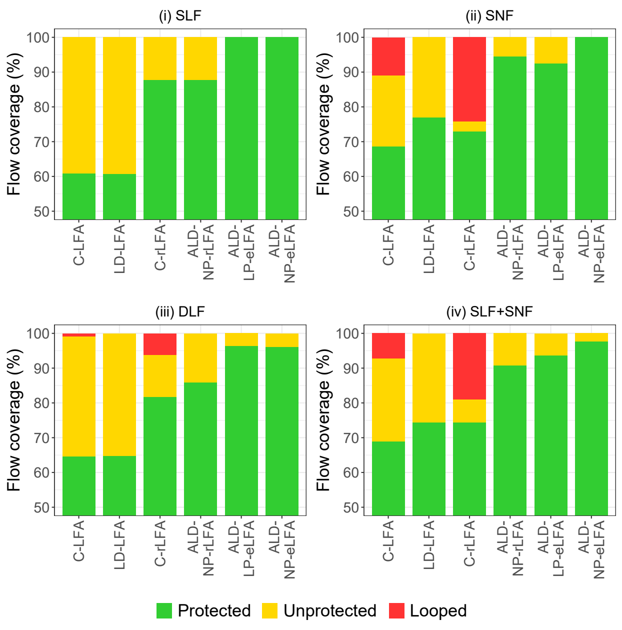
2) Evaluated Protection Variants: We consider the classic protection variants C-LFA (nLD-LP-LFA) and C-rLFA (nLD-LP-rLFA) as well as the LD-LFA (ALD-NP-LFA) from [1]. We further study the new protection variants ALD-NP-rLFA and ALD-{LP,NP}-eLFA since they have stronger protection properties.

3) Coverage: In this section we present results for the number of protected destinations for different failure scenarios. First, we evaluate unit link cost networks. Then, we discuss non-unit link cost networks.

a) Networks with Unit Link Costs: Figure Figure 3(a) shows the coverage in percent for different sets of failure scenarios in networks with unit link costs. Subfigure 3(a) (i) shows that only C-LFA and LD-LFA cannot protect all destinations against SLF, i.e., their coverage is less than 100%. All other protection variants provide full coverage.



(a) Networks with unit link costs.



(b) Networks with non-unit link costs.

Figure 3: Coverage averaged over 208 topologies depending on protection method and set of failure scenarios.

Subfigure 3(a) (ii) shows that SNFs cause many rerouting loops with C-LFA (17%) and C-rLFA (34%). This is mostly caused by failed destinations. As C-rLFA protect more destinations than C-LFA, they also cause more loops when the next-hop is the destination. Thus, loop detection is even more important when C-rLFA is used because more flows loop in case of node failures than with C-LFA. LD-LFA protects more traffic (81%) than C-(r)LFA in case of SNF as it preferentially uses NP-LFAs if available. Moreover, it prevents loops.

The new protection variants have significantly higher coverage. ALD-NP-rLFA protects around 99% of the destinations with SNF. This results from dropping packets that cannot be delivered anymore due to a failed destination; if they looped, the corresponding flow would count as looped. The coverage of ALD-LP-eLFA is slightly lower, i.e., 94%. This is because NP-(e/r)LFAs are not preferentially chosen for this protection variant so that there are more LFAs in use without the NP property. Finally, ALD-NP-eLFA protects all destinations for three reasons. First, it leverages rLFAs or eLFAs to provide protection for destinations that cannot be protected with LFAs. Second, it uses NP-(e/r)LFAs to protect against node failures and falls back to unsafe LP-(e/r)LFAs only when (e/r)LFAs with NP property are not available. Third, ALD detects and stops all loops that may be caused by LFAs with LP. This turns flows that cannot reach their destination into protected flows instead of looped flows.

Subfigure 3(a) (iii) shows the coverage against DLFs. No mechanism is able to protect all destinations. C-LFA and LD-LFA protect around 70% of the destinations. C-rLFA cover more flows (92%). However, protection variants without loop detection, i.e., C-LFA and C-rLFA, lead to loops. All newly proposed protection variants achieve roughly the same coverage, i.e., 96%, and prevent loops.

Finally, Subfigure 3(a) (iv) shows results for SLF+SNF. They are similar to the results of DLFs, but the fraction of rerouting loops caused by both C-LFA and C-rLFA is significantly higher. This is due to node failures which cause significant rerouting loops for protection variants without loop detection.

b) Networks with Non-Unit Link Costs: Figure Figure 3(b) shows the coverage for different sets of failure scenarios in networks with non-unit link costs. Subfigure 3(b) (i) shows the coverage against SLF. Both C-LFA and LD-LFA protect only around 60% of the destinations. In networks with non-unit link costs, C-rLFA cannot protect all destinations anymore against SLF and achieve only a coverage of 88%. The same holds for ALD-NP-rLFA. Only the eLFA-based protection variants are able to protect all destinations against SLF.

Subfigure 3(b) (ii) shows the coverage against SNF. Both C-LFA and C-rLFA cause many rerouting loops. LD-LFA prevents loops but protects only 76% of the destinations. ALD-NP-rLFA and ALD-LP-eLFA protect a higher fraction of destinations, i.e., 94% and 93%, because they prevent loops of unsafe LFAs with LP, but they have no suitable backup path for some node failures. ALD-NP-eLFA protects all destinations against SNF even in networks with non-unit link costs as it prevents loops and leverages NP-(e/r)LFAs wherever possible.

Finally, Subfigure 3(b) (iii) and Subfigure 3(b) (iv) present the coverage for DLF and SLF+SNF. The results are similar to those from networks with unit link costs, but the coverage here is slightly lower.

C. Additional Forwarding Entries

We now evaluate the number of additional forwarding entries to implement explicit tunnels. First, we explain the metric. Then, we discuss the investigated FRR mechanisms. Finally, we present results for networks with unit link costs and non-unit link costs.

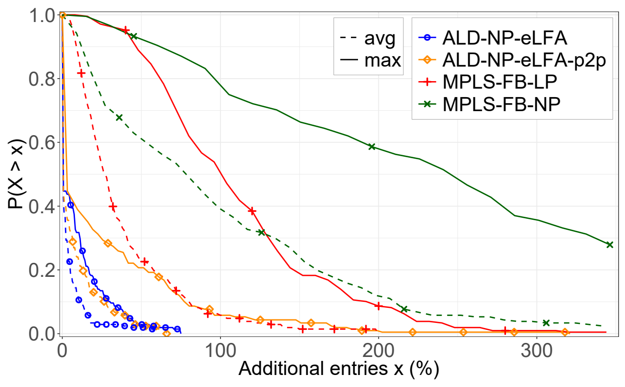
1) Metric: In a network with n nodes, each node maintains $n-1$ forwarding entries for destination-based forwarding. eLFAs require additional forwarding entries to implement explicit tunnels. In contrast, both LFAs and rLFAs are based on shortest paths, and therefore, do not need additional forwarding entries. We calculate the average and maximum amount of additional forwarding entries per node relative to $n-1$ for each network and present the results for all topologies in a CCDF.

2) FRR Mechanisms under Study: We compare the required amount of additional forwarding entries only for eLFA-based RoLPS protection variants as others do not require additional forwarding entries. To evaluate the efficiency of multipoint-to-point rerouting rerouting tunnels, we report results for ALD-{LP,NP}-eLFA and compare them to the corresponding mechanisms with point-to-point rerouting tunnels, i.e., ALD-{LP,NP}-eLFA-p2p. In addition, we present results for state-of-the-art MPLS-facility-backup (MPLS-FB-{LP,NP}) with LP and NP property.

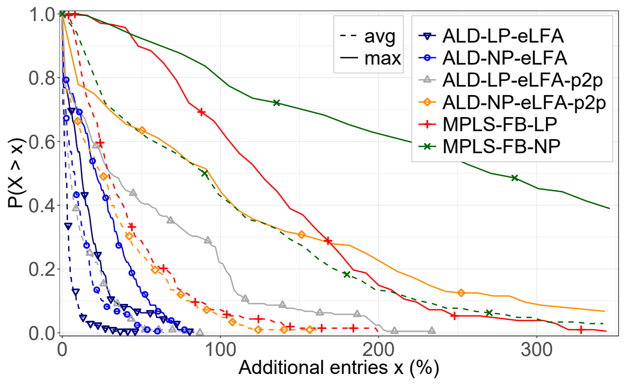
3) Results: We present results for the fraction of additional forwarding entries. First, we evaluate unit link cost networks. Then, we discuss non-unit link cost networks.

a) Networks with Unit Link Costs: Figure 4(a) shows a CCDF for the relative amount of additional forwarding entries for the considered FRR mechanisms in networks with non-unit link costs. First, we compare LP mechanisms. With MPLS-FB-LP, in 40% of the networks at least one node requires 120% or more additional entries (max-curve). However, on average in only 6% of the networks more than 100% additional entries are needed (avg-curve). The curves for ALD-LP-eLFA and ALD-LP-eLFA-p2p are omitted because those protection variants do not induce any additional forwarding entries. This is because (r)LFAs alone protect all destinations against all SLF in networks with unit link costs. Therefore, explicit LFAs are not needed and no additional forwarding entries are required.

Now, we compare NP mechanisms. MPLS-FB-NP requires most additional entries by far. 62% of the topologies have at least one node that requires 200% or more additional entries. And in 40% of the topologies 100% or more additional entries are required on average. Protection mechanisms with eLFAs, i.e., ALD-NP-eLFA and ALD-NP-eLFA-p2p, require less forwarding entries because they protect most of the destinations by NP-rLFAs and only the few remaining destinations are protected by eLFAs which induce forwarding state in the network. When ALD-NP-eLFA-p2p is used, only 20% of topologies have a node that requires 50% or more additional entries. However, some topologies contain at least one node that requires 200% or more additional entries. On average, no topology requires more than 65% or more additional entries. ALD-NP-eLFA is even more efficient because it leverages multipoint-to-point rerouting tunnels to reduce the number



(a) Networks with unit link costs. ALD-LP-eLFA does not induce any additional entries and is omitted in the figure.



(b) Networks with non-unit link costs.

Figure 4: CCDFs for fraction of additional forwarding entries.

of additional forwarding entries even further. There is no topology with a node that requires more than 70% of additional entries. 90% of the networks require only 15% or less additional entries on average.

b) Networks with Non-Unit Link Costs: Figure 4(b) shows a CCDF for the relative amount of additional forwarding entries for the considered FRR mechanisms in networks with non-unit link costs. Again, we compare LP mechanisms first. MPLS-FB-LP requires lots of additional entries. Around 55% of the topologies have at least one node that requires 120% or more additional entries (max-curve). However, in only 8% of the networks more than 100% additional entries are needed on average (avg-curve). eLFA-based protection mechanisms, i.e., ALD-LP-eLFA and ALD-LP-eLFA-p2p, are more efficient. When ALD-LP-eLFA-p2p is used, 22% of networks contain at least one node that requires 100% or more additional entries. On average, in 20% of networks nodes require 25% or more additional entries. ALD-LP-eLFA reduces the number of additional forwarding entries by leveraging multipoin-to-point tunnels. There is no topology with a node that requires more than 80% of additional entries and in 95% of the networks less than 15% additional entries are needed on average.

Now we compare NP mechanisms. MPLS-FB-NP requires most additional entries by far. 75% of networks have at least one node that requires 120% or more additional entries, 40% even more than 340%. In around 44% of the networks, 100% or more entries are required on average, and in 8% of the networks even 250% or more additional entries are required. ALD-NP-eLFA-p2p requires less entries. Only 45% of networks contain a node that requires 100% or more additional entries, but 20% of networks require even more than 210%. On average, 22% of networks require 50% or more additional entries. ALD-NP-eLFA is even more efficient. No network contains a node that requires more than 80% additional entries. In 90% of the networks, less than 30% additional entries are required on average.

Thus, in networks with non-unit link costs, somewhat more additional entries are needed but ALD-{LP,NP}-eLFA still require significantly less entries than MPLS-FB-{LP,NP} and ALD-{LP,NP}-eLFA-p2p.

D. Size of Header Stacks for TI-LFAs

In this section we evaluate the number of required segments to implement explicit paths with TI-LFAs using local labels. First, we explain the metric, and the studied mechanisms. Then, we present the results.

1) Metric: We count the number of header segments that are added to the packet by the respective mechanism for FRR purposes. That is, we do not count the header segment that identifies the original destination of the packet. For each network we record both the average and maximum number of additional header segments added to a packet and present the results as a CCDF.

2) Reroute Mechanisms under Study: We evaluate TI-LFAs that use only local labels (see Section III-C) because they implement explicit tunnels with multiple header segments. However, we leverage TI-LFAs only when there are no LFAs or rLFAs to protect a destination to avoid unnecessary additional header segments.

rLFAs, and eLFAs also leverage tunnels. However, both require only one additional header segment for tunneling which is why we omit those curves in the figure to facilitate readability. LFAs do not require additional header segments.

3) Results: Figure 5 shows the results for networks with non-unit link costs.

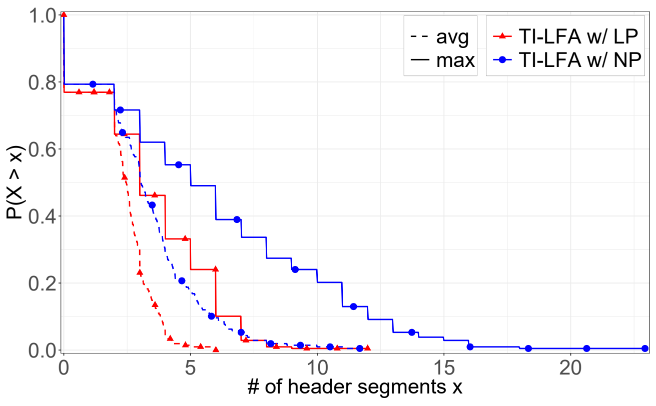


Figure 5: CCDF for number of additional header segments.

First, we discuss TI-LFAs with LP. In most networks, i.e., roughly 80%, TI-LFAs with LP require at least two additional header segments. In 20% of networks TI-LFAs with LP require on average 3 or more additional header segments. However, 23% of networks have at least one TI-LFA with LP that requires 6 or more additional header segments.

Now, we discuss TI-LFAs with NP. In general, TI-LFAs with NP require more additional header segments than TI-LFAs with LP. In 19% of networks TI-LFAs with NP require on average 5 or more additional header segments. 40% of networks even contain at least one TI-LFA with NP that requires 6 or more additional header segments.

In Section III-C2 we mentioned that the size of the TI-LFA header stack may be reduced. This, however, requires optimization which is a promising approach and an interesting research issue, but it is out of the scope of this document.

We omit a figure for results for networks with unit-link costs because of two reasons. First, TI-LFAs with NP require slightly fewer header segments but the results show no further insights. Second, for LP all destinations can be protected with either LFAs or rLFAs, i.e., no TI-LFAs are used, which was to be expected.

E. Path Lengths

In this section we report results for path lengths. First, we explain the metric and evaluated FRR mechanisms, then, we present the results.

1) Metric: We measure the path lengths of all flows that are affected by SLF but were successfully delivered due to local rerouting. For each topology, we calculate the average and maximum path lengths and present the results for all topologies in a CCDF.

2) Reroute Mechanisms under Study: We choose path lengths for rerouting as a baseline which recomputes shortest paths after a failure. We compare these results to the ones for ALD-{LP,NP}-eLFA and MPLS-FB-{LP,NP}.

3) Results: Figure 6 shows a CCDF for average and maximum path lengths of successfully delivered flows with SLF in networks with unit link costs.

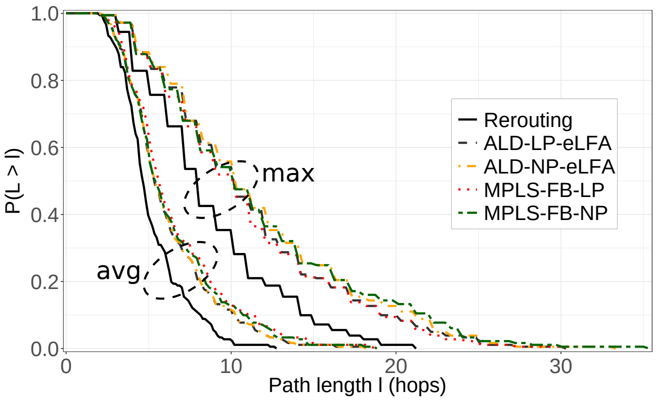


Figure 6: CCDF for path lengths of successfully delivered flows for SLF in networks with unit link costs.

We observe that rerouting leads in fact to the shortest maximum and average path lengths. All FRR mechanisms under study lead to longer maximum and average path lengths. The path lengths of the different FRR mechanisms does not differ.

The same analysis in networks with non-unit link costs leads to slightly longer paths but without any further insights. Therefore, we omit the corresponding figure.

F. Discussion

We investigated various RoLPS protection variants with regard to protection coverage, additional forwarding entries, and path lengths on a set of 208 topologies with both unit link costs and non-unit link costs, and compared them with MPLS-facility-backup.

The evaluations of protection coverage showed that C-LFA cannot protect many destinations in case of link failures. C-rLFAs can protect all destinations in case of SLF in networks with unit link costs. However, the usage of C-(r)LFA leads to many loops in case of node failures. The use of ALD avoids such loops. LD-LFA [1] prevents loops but cannot protect all destinations. ALD-NP-eLFA protects all destinations against SLF and SNF in networks with unit and non-unit link costs because it leverages eLFAs to complement (r)LFAs.

The explicit LFAs induce additional forwarding entries in the data plane, which is not desired. Therefore, we compared the additional forwarding entries for ALD-{LP,NP}-eLFA, ALD-{LP,NP}-eLFA-p2p, and MPLS-FB-{LP,NP}. ALD-{LP,NP}-eLFA require only very few additional entries compared to ALD-{LP,NP}-eLFA-p2p, and MPLS facility backup. Both MRCs [14] and IDAGs [18] always require 100% additional entries, and MRTs [15] need 200% more. Not-via addresses [11] need $100\% \cdot d$ more entries where d is the average node degree. Although TI-LFAs require at most d additional forwarding entries per node, they impose significant overhead in form of multiple additional header segments. ALD-{LP,NP}-eLFA add only one additional packet header for tunneling and our evaluation shows that they require less additional forwarding entries than other comparable FRR mechanisms. Therefore, ALD-{LP,NP}-eLFA can be considered very lightweight which makes them attractive for FRR in SDN.

All evaluated FRR mechanisms, i.e., ALD-{LP,NP}-eLFA and MPLS-FB-{LP,NP} extend backup paths by about the same, and backup paths are only slightly longer than the average and maximum length of recomputed shortest paths.

VI. Implementation of RoLPS in P4

We start with a short introduction of P4 and the implementation platform. Then we summarize important basics of P4 and describe the implementation of the RoLPS prototype.

A. Overview of P4 and the Implementation Target

P4 is a high-level programming language for protocol-independent packet processors [51]. P4 programs are mapped, i.e., compiled, to the programmable processing pipeline of so-called targets, e.g., the software switch BMv2 [52] or the switching ASIC Tofino [53]. When a P4 program is successfully compiled for a target, it offers an API to let the control plane configure the device during runtime, e.g., to write forwarding entries.

In [2] we sketched how the predecessor of RoLPS could be implemented in OpenFlow. However, due to technical restrictions of OpenFlow the implementation concept required multiple workarounds which made it complex (see Section III-B1 and Section IV-C1). P4 offers significantly more flexibility than OpenFlow. It allows a flexible description of the data plane, in particular, the definition of arbitrary packet headers and packet parsers, and conditional application of programmable match+action tables (MATs). Therefore, implementation of novel features in P4 is easier than in OpenFlow.

In this paper we describe the implementation of RoLPS in P4. Our target is the P4-programmable high-performance switching ASIC Tofino [53] which is used in the Edgecore Wedge 100BF-32X [54] switch with 32 100 Gb/s ports. We made the source code for the RoLPS data plane and control plane publicly available[2].

B. P4 Pipeline

Figure 7 illustrates the abstract forwarding model of P4. A user-programmable parser extracts the information from the packet header and stores them in so-called header fields. They are carried with the packet through the processing pipeline, possibly with additional metadata which are similar to regular variables from other high-level programming languages. Metadata are packet-specific and discarded after the packet is sent to an egress port.

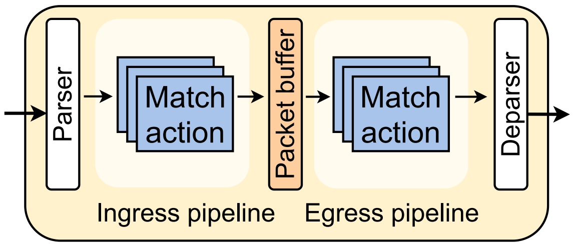


Figure 7: P4 abstract forwarding model according to [51].

The P4 abstract forwarding model is divided into two stages, the ingress and the egress pipeline, which are separated by a packet buffer. Match+action tables (MATs) allow for packet-specific processing. They have entries consisting of custom match fields and types that map header fields and metadata to actions, e.g., modifying header fields, and parameters.

P4 offers three match types: exact, longest-prefix match (LPM), and ternary. For an exact match the header field or metadata field must be exactly the same as the match field in the MAT, e.g., a specific IP address. LPM is well-known from standard IP forwarding. Ternary facilitates wildcard matches. P4 does not allow to match a packet multiple times on the same MAT to prevent processing loops.

[2]https://github.com/uni-tue-kn/p4-lfa

After the egress pipeline, the deparser writes the potentially modified header fields into the packet header and the packet is sent through the specified egress port.

However, P4 does not support FRR natively. Port status information cannot be accessed by the data plane by default. This makes the implementation of FRR in P4 a serious challenge.

C. Implementation of LFAs

First, we describe how the port status can be determined in P4. Afterwards, we describe the implementation of LFAs without tunnels followed by LFAs with tunnels, i.e., rLFAs and eLFAs, and ranking-based selection of LFA types.

1) Port Status Detection in P4: Executing backup actions, e.g., forwarding to an LFA, requires a reliable and timely detection when a port goes down. However, P4 does not support such a feature. In [55] we proposed a workaround for the Tofino platform which detects port-down events within 1 ms without controller interaction. We leverage this workaround to implement RoLPS-based protection and summarize it in the following.

Registers in P4 provide persistent storage, i.e., their content survives processed packets. The individual register fields can be accessed by an index. We leverage a register to store the current status of the egress ports by single bits (0: down, 1: up). Each register field stores the status of one port, i.e., one bit. The port ID serves as an index to access the corresponding register field. The challenge is updating the registers when the port status changes, which is platform-specific.

Port-down events are tracked as follows. Tofino has means outside the P4 programmable data plane to detect port-down events. We configured the Tofino such that it creates a ‘port-down packet’ in case of a port-down event. The packet contains the ID of the corresponding port and the packet is sent to a switch-intern port. We programmed the p4 pipeline such that the port status register for the respective port is set to zero upon reception of a port-down packet.

Port-up events are tracked differently. When the Tofino receives a packet over a specific port, it activates the status bit of that port in the register. To ensure that port-up events are detected sufficiently fast, we take advantage of topology packets that are regularly sent by the Tofino to all egress ports for neighbor detection. The frequency for topology packets can be configured to an appropriate value. While the detection of port-down events is time-critical, detection of port-up events is more relaxed because FRR mechanisms reroute affected traffic in the meantime via alternative ports.

2) Implementation of LFAs without Tunnels: As described in the previous section, the register fields provide information whether specific egress ports are up or down. However, the egress port of a packet is known only after matching the packet on a MAT. To mitigate this problem, we implemented FRR as shown in Figure 8. First, the packet is matched against a MAT that performs regular IPv4 routing, i.e., it determines the next-hop and thereby the egress port of a packet. Second, the ID of the selected egress port is used to access the register fields to retrieve the port status of that egress port. If the egress port is

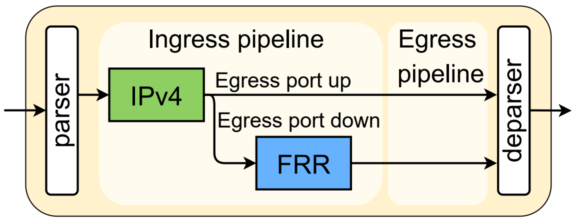


Figure 8: P4 implementation of FRR. A packet is matched against an IPv4 forwarding MAT to determine its egress port. If that port is down, the packet is matched against a FRR-MAT to determine its backup egress port.

up, the packet is forwarded. If the port is down, FRR actions are triggered, i.e., the packet is matched against a FRR-MAT using the IP destination address and the ID of the failed egress port. This selects a backup entry with a preinstalled LFA, i.e., backup egress port, for forwarding.

3) LFAs with Tunnels: LFAs with tunnels are implemented in a similar way as LFAs without tunnels. However, the backup actions in the FRR-MAT contain an encapsulation action which adds an additional IP header to the packet for tunneling to the remote node, i.e., the rLFA or eLFA.

If the remote node is an rLFA, the encapsulating IP header contains the IP address of that node. The packet is then forwarded on standard paths towards the rLFA.

If the remote node is an eLFA, the encapsulating IP header contains a unique IP address which identifies the explicit path towards the eLFA (see Section IV-B2). When the controller installs eLFAs in the network, it also sets up explicit tunnels towards the eLFAs. To that end, it calculates appropriate tunnel-specific forwarding entries and configures them on the forwarding devices along the explicit path. Thereby, the controller leverages explicit multipoint-to-point rerouting tunnels (see Section IV-B3) if possible to reduce the number of additional forwarding entries. That is, it configures only one additional forwarding entry on forwarding devices on overlapping subpaths of explicit paths towards the same eLFA.

4) Implementation of Ranking-Based Selection of LFA Types: The ranking-based selection of LFAs as described in Section IV-D is part of the control plane. The controller precomputes appropriate LFA types depending on the desired protection variant and installs corresponding egress ports and encapsulation actions in the FRR-MATs of the data plane devices.

D. *Implementation of ALD*

We implement ALD so that it allows two redirects, i.e., the packet is dropped when it has to be rerouted a third time. To that end, we define the ALD field as a 2-bit custom header field in the packet header. These bits track how often a packet has been rerouted. Packets initially carry the bit pattern ‘00’ in the ALD field. When a node reroutes a packet with bit pattern ‘00’, it replaces the bit pattern with ‘01’. When a node reroutes a packet with bit pattern ‘01’, it replaces the bit pattern with ‘10’. When a node cannot forward a packet with bit pattern ‘10’ due to a failed egress port, it drops the packet.

VII. Hardware-Based Performance Evaluation

In this section we conduct a performance evaluation of the RoLPS hardware prototype. It is based on the Tofino [53], a P4-programmable switch ASIC, which is used in the Edgecore Wedge 100BF-32X [54], a switch with 32 100 Gb/s ports. We present measurement results for throughput, restoration time, and loop detection.

A. *Throughput*

Every P4 program successfully compiled for the Tofino processes packets at a speed of 100 Gb/s. To verify that property for our prototype, we conducted the following experiment. We utilized an EXFO FTB-1 Pro traffic generator [56] which generates up to 100 Gb/s of traffic. We connected it to the Tofino which processes the traffic and sends it back to the traffic generator. This way we measure the traffic rate forwarded by Tofino. In fact, we obtained a throughput of 100 Gb/s for both failure-free forwarding and forwarding with activated FRR.

B. *Restoration Time*

The evaluation of restoration times is more complex. We describe the testbed, the measurement procedure and metric, as well as the experimental scenarios. Then, we present measurement results.

1) Testbed: Figure 9 shows the testbed for the performance evaluation. Center of the testbed is the above mentioned Tofino.

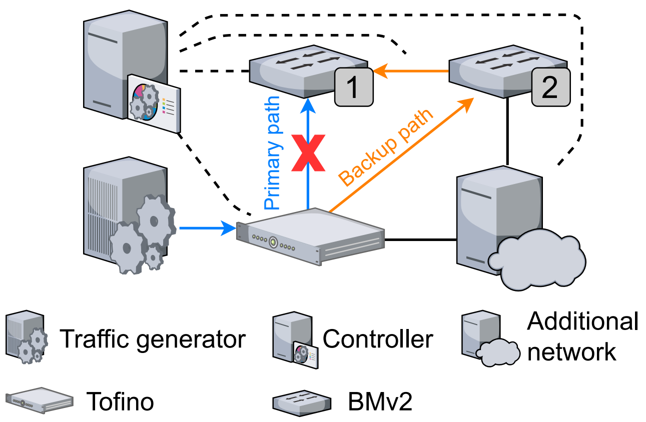


Figure 9: Topology for restoration time measurements. The additional network consists of five other BMv2s and 10 links.

It is connected to two BMv2 [52] P4 software switches. To perform evaluations for more realistic network sizes, we connected the Tofino to an additional network which consists of five BMv2s and 10 links. All BMv2s run on a server with an Intel Xeon Gold 6134 with 3.2 GHz and 12 cores, and 32 GB RAM. A controller is connected to the Tofino and all BMv2s. It configures them upon start, i.e., it discovers the topology, and computes and installs appropriate forwarding rules. It runs on the same server as the BMv2s. Furthermore, the above mentioned traffic generator is connected to the Tofino and serves as a traffic source in the experiment.

2) Measurement Procedure and Metric: The traffic generator sends traffic to the Tofino which forwards the packets on the primary path to the destination BMv2-1. BMv2-1 monitors the packet arrivals. Then, we deactivate the link from Tofino to BMv2-1 on the primary path to trigger a port-down event at the Tofino. We derive the restoration time for the FRR mechanism from a tcpdump log at BMv2-1. It is the duration of the interval within which BMv2-1 does not receive any packets.

In these experiments, the traffic generator sends only with 100 Mb/s instead of 100 Gb/s. This avoids overload on the BMv2s which can process packets only with around 900 Mb/s [57]. Avoiding overload is important only to obtain correct measurement results from BMv2-1. The restoration time on the Tofino is not affected by any overload.

3) Experiments: We perform two experiments to measure the restoration time without and with FRR.

a) Forwarding without FRR: For this experiment we disabled the FRR feature on Tofino. When the Tofino detects the failure, it notifies the controller. The controller then updates its topology, computes new forwarding entries, and installs them on the affected devices so that traffic can be forwarded again.

b) Forwarding with FRR: In this experiment the FRR feature is enabled. Thus, if BMv2-1 is no longer reachable, the Tofino forwards traffic destined to BMv2-1 to BMv2-2 which relays the traffic to BMv2-1.

4) Results: We performed the above described experiments 10 times. Figure 10 shows the average restoration time without and with FRR on the Tofino, including 95% confidence intervals.

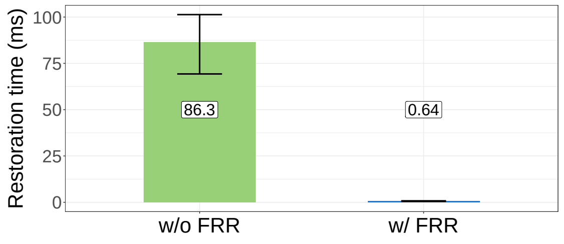


Figure 10: Restoration time on Tofino without and with FRR.

If FRR is disabled, traffic is delivered again after 86 ms. As rerouting without FRR requires controller interaction, the measured restoration time depends on controller load, network size, and communication delay. In this experiment, there is only a single flow affected by the faiure, the overall network is small despite the additional network, and the controller is directly connected to the Tofino. Therefore, the experimental result for the restoration time is likely lower than restoration times in production networks.

If FRR is enabled, traffic is delivered after a small restoration time of 0.6 ms. Here, the switchover from primary egress port to backup egress port at the Tofino is independent of controller load, network size, and communication delay as FRR is a switch-local mechanism. Thus, restoration times can be greatly reduced by FRR on P4-capable hardware. Moreover, the mechanism is general enough to support all RoLPS protection variants by appropriate configuration through the controller.

C. Loop Detection

We experimentally evaluate the capability of ALD to detect and stop loops. We present the modified testbed, explain two different experiments and the studied metric, and finally we discuss measurement results.

1) Testbed: Figure 11 shows the testbed. The Tofino is

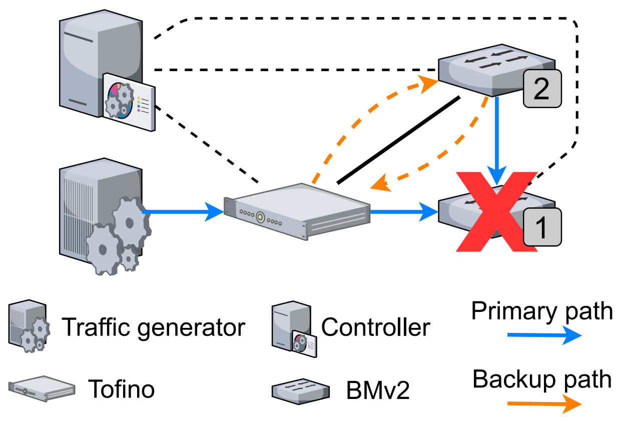


Figure 11: Testbed for evaluation of ALD.

now conntected to two BMv2s (BMv2-1, BMv2-2) which are also connected with each other. The controller configures the Tofino and all BMv2s with available LP-LFAs upon startup. In the experiments, the traffic generator sends a packet towards BMv2-1. The Tofino has BMv2-2 as an LFA when BMv2-1 is not reachable. Likewise, BMv2-2 has the Tofino as an LFA when BMv2-1 is not reachable. If BMv2-1 fails, traffic destined to that node loops between the Tofino and BMv2-2. However, the TTL in the IP header is set to 64 when sent by the traffic generator and decremented whenever forwarded by a node. The packet is dropped when its TTL reached 0.

2) Experiments and Metric: We perform two experiments with ALD disabled and ALD enabled on the switches. We track packet arrivals at BMv2-2 using tcpdump. Thereby we can observe how often a looping packet is received.

3) Results: Figure 12 illustrates a log of packet arrivals at BMv2-2, starting with time 0 at first packet arrival. Without

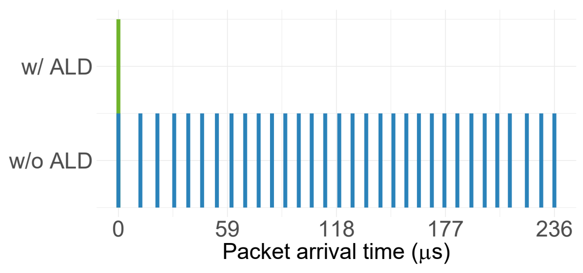


Figure 12: Packet arrivals at BMv2-2 without and with ALD.

ALD, BMv2-2 receives the packet 32 times. Thus, the packet looped between the Tofino and BMv2-2 until it was dropped due to TTL=0. With ALD, BMv2-2 receives the packet only once. It then redirects the packet to the Tofino which then drops the packet at the attempt to reroute the packet for the third time. Therefore, BMv2-2 receives the packet only once.

VIII. Conclusion

In this paper we presented robust LFA protection for software-defined networks (RoLPS). It leverages loop-free alternates (LFAs) and remote LFAs (rLFAs) known from IP networks to forward traffic over alternative next-hops if primary next-hops are not reachable. However, this alone cannot protect all destinations against failures and may cause forwarding loops under challenging conditions. Therefore, we proposed explicit LFAs (eLFAs) using explicit rerouting tunnels to cover all destinations. eLFAs are conceptually similar to topology-independent LFAs (TI-LFAs) but do require only a single additional header segment for protection while protection with typical TI-LFAs may require a clearly larger header stack. Furthermore we describe advanced loop detection (ALD) to stop forwarding loops. These mechanisms are simple and do not require controller interaction. We suggested various protection variants that utilize (e/r)LFAs with different protection quality and complexity.

We evaluated RoLPS through simulations based on 208 representative topologies. The results revealed that existing (r)LFAs cannot provide all destinations and lead to substantial forwarding loops in case of node failures. More elaborate RoLPS variants with eLFAs and ALD, e.g., ALD-NP-eLFA, protect all traffic against all single link or node failures in networks with both unit and non-unit link costs. Furthermore, they protect most destinations against multiple failures ($> 90\%$) and prevent forwarding loops. A drawback of eLFAs is that they required additional forwarding entries. However, our evaluation showed that RoLPS protection variants require only very few eLFAs, in particular compared to other FRR mechanisms such as MPLS facility backup, MRTs, MRCs, IDAGs, or not-via addresses. Thus, the full protection coverage against single link or node failures together with the need for only a few additional forwarding entries make RoLPS attractive for software-defined networks. In addition, RoLPS protection variants extends lengths of backup paths compared to those of shortest path recomputation, but there is no visible difference to backup path lengths with MPLS facility backup.

We implemented a P4-based prototype that features RoLPS-based protection variants. The source code is publicly available. A measurement study showed that the prototype achieves a throughput of 100 Gb/s, restores connectivity in less than 1 ms including failure detection, and reliably detects and stops forwarding loops.

Acknowledgment

We acknowledge the support from BelWü for borrowed high-performance hardware for the measurement-based evaluations. Likewise, we appreciate the work of Irene Müller-Benz for the development of an early prototype of RoLPS.

Acronyms and Glossary

FRR	fast reroute
PLR	point of local repair
LFA	loop-free alternate [20]
rLFA	remote LFA [23], [24]
eLFA	explicit LFA [2]
TI-LFA	topology-independent LFA [3]
MPLS	multiprotocol label switching [8]
MRT	maximally redundant tree [15]
IDAG	independent directed acyclic graph [18]
MRC	multiple routing configuration [14]
SLF	single link failure
SNF	single node failure
DLF	double link failure
LP	link protecting
NP	node protecting
ALD	advanced loop detection
RoLPS	robust LFA protection for SDN

Table 3: Acronyms.

Point of local repair (PLR)	A node that cannot forward a packet to the default next-hop because of a failure. It executes precomputed backup actions to locally reroute packets around the failure.
Loop-free alternate (LFA)	Alternative next-hop that successfully forwards failure-affected traffic towards the destination. Simple LFAs cannot protect all destinations.
rLFA	Remote nodes in the network that successfully forward traffic towards the destination. PLRs reach rLFAs through shortest path tunnels. rLFAs protect more destinations than LFAs. However, they cannot protect all destinations against SLF in non-unit link cost networks or SNF in general.
eLFA	Similar to rLFAs. However, PLRs reach eLFAs through explicit tunnels implemented by additional forwarding entries. eLFAs protect against all SLF and SNF independent of link costs. Multipoint-to-point tunnels reduce the number of additional forwarding entries.
Link protecting (LP)	A link protecting (e/r)LFA avoids the link between PLR and next-hop. They may cause rerouting loops for SNF.
Node protecting (NP)	A node protecting (e/r)LFA avoids the next-hop. There are significantly less NP-(e/r)LFAs than LP-(e/r)LFAs. NP implies LP, i.e., it is the stronger property.
Loop detection (LD) [1]	A mechanism to detect and stop rerouting loops caused by LFAs. May erroneously drop packets.
LD-LFA [1]	LD-LFA preferably uses NP-LFAs for protection. Only when no NP-LFA is available, LP-LFAs are used to increase the number of protected destinations. In addition, LD-LFA leverages loop detection to prevent loops.
Advanced loop detection (ALD)	A mechanism to detect and stop loops caused by LFAs. Allows to reroute a packet two times to cope with double failures.
Robust LFA protection for SDN (RoLPS)	Protection concept presented in this paper. It defines eLFAs and ALD. RoLPS ranks (e/r)LFAs and selects the best one. Uses ALD to detect and stop loops.

Table 4: Glossary.

References

[1] W. Braun and M. Menth, "Loop-Free Alternates with Loop Detection for Fast Reroute in Software-Defined Carrier and Data Center Networks," *Journal of Network and Systems Management*, vol. 24, 2016.

[2] D. Merling, W. Braun, and M. Menth, "Efficient Data Plane Protection for SDN," in *IEEE Conference on Network Softwarization (NetSoft)*, Jun. 2018.

[3] P. Francois, C. Filsfils, A. Bashandy, B. Decraene, and S. Litkowski, *Topology Independent Fast Reroute using Segment Routing*, https://tools.ietf.org/html/draft-ietf-rtgwg-segment-routing-ti-lfa-06, Feb. 2021.

[4] S. Rai, B. Mukherjee, and O. Deshpande, "IP Resilience within an Autonomous System: Current Approaches, Challenges, and Future Directions," *IEEE Communications Magazine*, vol. 43, 2005.

[5] A. Raj and O. Ibe, "A Survey of IP and Multiprotocol Label Switching Fast Reroute Schemes," *Computer Networks*, vol. 51, no. 8, 2007.

[6] J. Papan, P. Segeč, P. Palúch, and L. Mikus, "The Survey of Current IPFRR Mechanisms," in *Federated Conference on Software Development and Object Technologies*, Dec. 2017.

[7] D. Hutchison and J. P. Sterbenz, "Architecture and design for resilient networked systems," *Computer Communications*, vol. 131, 2018.

[8] E. Rosen, A. Viswanathan, and R. Callon, *Multiprotocol Label Switching Architecture*, https://tools.ietf.org/html/rfc3031, Jan. 2001.

[9] Ping Pan and George Swallow and Alia Atlas, *RFC4090: Fast Reroute Extensions to RSVP-TE for LSP Tunnels*, https://tools.ietf.org/html/rfc4090, May 2005.

[10] K. Kompella and W. Lin, *No Further Fast Reroute*, https://tools.ietf.org/html/draft-kompella-mpls-nffrr-00, Mar. 2020.

[11] S. Bryant, S. Previdi, and M. Shand, *RFC6981: A Framework for IP and MPLS Fast Reroute Using Not-Via Addresses*, http://www.rfc-editor.org/rfc/rfc6981.txt, Jul. 2013.

[12] R. Martin, M. Menth, M. Hartmann, T. Cicic, and A. Kvalbein, "Loop-Free Alternates and Not-Via Addresses: A Proper Combination for IP Fast Reroute?" *Computer Networks*, vol. 54, 2010.

[13] S. Nelakuditi *et al.*, "Fast Local Rerouting for Handling Transient Link Failures," *IEEE/ACM Trans. on Networking*, Apr. 2007.

[14] A. Kvalbein, A. F. Hansen, T. Cicic, S. Gjessing, and O. Lysne, "Fast IP Network Recovery Using Multiple Routing Configurations," in *IEEE Infocom*, Apr. 2006.

[15] A. Atlas, C. Bowers, and G. Enyedi, *RFC7812: An Architecture for IP/LDP Fast Reroute Using Maximally Redundant Trees (MRT-FRR)*, http://www.rfc-editor.org/rfc/rfc7812.txt, Jun. 2016.

[16] M. Menth and W. Braun, "Performance Comparison of Not-Via Addresses and Maximally Redundant Trees (MRTs)," in *IEEE/IFIP IM*, Apr. 2013.

[17] K. Kuang, S. Wang, and X. Wang, "Discussion on the Combination of Loop-Free Alternates and Maximally Redundant Trees for IP Networks Fast Reroute," in *IEEE International Conference on Communications*, Jun. 2014.

[18] S. Cho, T. Elhourani, and S. Ramasubramanian, "Independent Directed Acyclic Graphs for Resilient Multipath Routing," *IEEE/ACM Transactions on Networking*, vol. 20, Feb. 2012.

[19] S. S. Lor, R. Landa, and M. Rio, "Packet re-cycling: Eliminating packet losses due to network failures," in *ACM Workshop on Hot Topics in Networks*, 2010.

[20] A. Atlas and A. Zinin, *RFC5286: Basic Specification for IP Fast Reroute: Loop-Free Alternates*, http://www.rfc-editor.org/rfc/rfc5286.txt, 2008.

[21] L. Csikor, M. Nagy, and G. Rétvári, "Network Optimization Techniques for Improving Fast IP-level Resilience with Loop-Free Alternates," *Infocommunications Journal*, vol. 3, 2011.

[22] L. Csikor, J. Tapolcai, and G. Retvari, "Optimizing IGP link costs for improving IP-level resilience with Loop-Free Alternates," *Computer Communications*, vol. 36, 2013.

[23] S. Bryant, C. Filsfils, S. Previdi, M. Shand, and N. So, *RFC7490:Remote Loop-Free Alternate (LFA) Fast Reroute (FRR)*, https://tools.ietf.org/html/rfc7490, 2015.

[24] P. Sarkar, S. Hegde, C. Bowers, H. Gredler, and S. Litkowski, *Remote-LFA Node Protection and Manageability*, https://tools.ietf.org/html/rfc8102, 2017.

[25] L. Csikor and G. Retvari, "On Providing Fast Protection with Remote Loop-Free Alternates: Analyzing and Optimizing Unit Cost Networks," in *Telecommunication Systems*, 2015.

[26] G. Retvari, J. Tapolcai, G. Enyedi, and A. Csaszar, "IP Fast ReRoute: Loop Free Alternates Revisited," in *IEEE Infocom*, Apr. 2011.

[27] W. Tavernier, D. Papadimitriou, D. Colle, M. Pickavet, and P. Demeester, "Self-configuring Loop-free Alternates with High Link Failure Coverage," *Telecommunication Systems*, vol. 56, 2014.

[28] A. Farrel and R. Bonica, "Segment Routing: Cutting Through the Hype and Finding the IETF's Innovative Nugget of Gold," *IETF Journal*, vol. 13, 2017.

[29] Y. E. Oktian *et al.*, "Distributed SDN Controller System: A Survey on Design Choice," *Computer Networks*, vol. 121, 2017.

[30] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "OpenFlow: Meeting Carrier-Grade Recovery Requirements," *Computer Communications*, vol. 36, 2013.

[31] A. S. da Silva, P. Smith, A. Mauthe, and A. Schaeffer-Filho, "Resilience support in software-defined networking: A survey," *Computer Networks*, vol. 92, 2015.

[32] M. Chiesa, A. Kamisiński, J. Rak, G. Rétvári, and S. Schmid, *A Survey of Fast Recovery Mechanisms in the Data Plane*, https://www.techrxiv.org/articles/preprint/Fast_Recovery_Mechanisms_in_the_Data_Plane/12367508/2, May 2020.

[33] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takàcs, and P. Sköldström, "Scalable Fault Management for OpenFlow," in *IEEE International Conference on Communications*, 2012.

[34] N. L. van Adrichem, B. J. van Asten, and F. A. Kuipers, "Fast Recovery in Software-Defined Networks," in *European Workshop on Software Defined Networks*, Sep. 2014.

[35] R. M. Ramos *et al.*, "SlickFlow: Resilient Source Routing in Data Center Networks Unlocked by OpenFlow," in *IEEE Conference on Local Computer Networks*, Oct. 2013.

[36] C. Cascone, L. Pollini, D. Sanvito, A. Capone, and B. Sansó, "SPIDER: Fault Resilient SDN Pipeline with Recovery Delay Guarantees," in *IEEE Conference on Network Softwarization*, Jun. 2016.

[37] N. L. M. van Adrichem, F. Iqbal, and F. A. Kuipers, "Backup Rules in Software-Defined Networks," in *IEEE Conference on Network Function Virtualization and Software-Defined Networking*, Nov. 2016.

[38] S. Cevher, M. Ulutas, S. Altun, and I. Hokelek, "Multi Topology Routing Based IP Fast Re-Route for Software Defined Networks," in *IEEE Symposium on Computers and Communications*, Jun. 2016.

[39] S. Cevher, "Multi Topology Routing Based Failure Protection for Software Defined Networks," in *IEEE International Black Sea Conference on Communications and Networking*, Jun. 2018.

[40] Q. Li, Y. Liu, Z. Zhu, H. Li, and Y. Jiang, "BOND: Flexible failure recovery in software defined networks," *Computer Networks*, vol. 149, 2019.

[41] R. Sedar, M. Borokhovich, M. Chiesa, G. Antichi, and S. Schmid, "Supporting Emerging Applications With Low-Latency Failover in P4," in *Workshop on Networking for Emerging Applications and Technologies*, 2018.

[42] H. Giesen, L. Shi, J. Sonchack, A. Chelluri, N. Prabhu, N. Sultana, L. Kant, A. J. McAuley, A. Poylisher, A. DeHon, and B. T. Loo, "In-Network Computing to the Rescue of Faulty Links," in *Morning Workshop on In-Network Computing*, 2018.

[43] S. Lindner, D. Merling, M. Häberle, and M. Menth, "P4-Protect: 1+1 Path Protection for P4," *P4 Workshop in Europe (EuroP4)*, Dec. 2020.

[44] K. Hirata and T. Tachibana, "Implementation of Multiple Routing Configurations on Software-Defined Networks with P4," in *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, 2019.

[45] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, *D2R: Dataplane-Only Policy-Compliant Routing Under Failures*, 2019.

[46] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisiński, G. Nikolaidis, and S. Schmid, "PURR: A Primitive for Reconfigurable Fast Reroute," in *ACM Conference on emerging Networking EXperiments and Technologies*, 2019.

[47] C. Filsfils *et al.*, *RFC8986: Segment Routing over IPv6 (SRv6) Network Programming*, https://www.rfc-editor.org/rfc/rfc8986.txt, 2021.

[48] A. Bashandy *et al.*, *RFC8660: Segment Routing with the MPLS Data Plane*, https://www.rfc-editor.org/rfc/rfc8660.txt, 2019.

[49] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The Internet Topology Zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, 2011.

[50] S. Halabi, *OSPF DESIGN GUIDE*, http://rtfm.vtt.net/spf1euro.pdf, 1996.

[51] P. Bosshart *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM CCR*, vol. 44, 2014.

[52] p4lang, *Behavioral-model*, https://github.com/p4lang/behavioral-model, 2019.

[53] Edge-Core Networks, *The World's Fastest & Most Programmable Networks*, https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/, 2017.

[54] ——, *Wedge100BF-32X/65X Switch*, https://www.edge-core.com/_upload/images/Wedge100BF-32X_65X_DS_R05_20191210.pdf, 2019.

[55] D. Merling, S. Lindner, and M. Menth, "Hardware-Based Evaluation of Scalable andResilient Multicast with BIER in P4," *IEEE Transactions on Network and Service Management*, In Revision for TNSM special issue: Advanced Management of Softwarized Networks.

[56] EXFO, *FTB-1v2/FTB-1 Pro Platform*, https://www.exfo.com/umbraco/surface/file/download/?ni=10900&cn=en-US&pi=5404, 2019.

[57] A. Bas, *BMv2 Throughput*, https://github.com/p4lang/behavioral-model/issues/537#issuecomment-360537441, Jan. 2018.

Daniel Merling is a Ph. D. student at the chair of communication networks of Prof. Dr. habil. Michael Menth at the Eberhard Karls University Tuebingen, Germany. There he obtained his master's degree in 2017 and afterwards, became part of the communication networks research group. His area of expertise include software-defined networking, scalability, P4, routing and resilience issues, multicast and congestion management.

Steffen Lindner is a Ph.D. student at the Eberhard Karls University Tübingen, Germany. He wrote his bachelor and master thesis at the chair of communication networks of Prof. Dr. habil. Michael Menth. He started his Ph.D. in September 2019 at the communication networks research group. His research interests include software-defined networking, P4 and congestion management.

Michael Menth (Senior Member, IEEE) is professor at the Department of Computer Science at the University of Tuebingen/Germany since 2010 and chairholder of Communication Networks. He studied, worked, and obtained diploma (1998), PhD (2004), and habilitation (2010) degrees at the universities of Austin/Texas, Ulm/Germany, and Wuerzburg/Germany. His special interests are performance analysis and optimization of communication networks, resilience and routing issues, resource and congestion management, industrial networking and Internet of Things, software-defined networking and Internet protocols.

2.6 A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research

A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research

Frederik Hauser[a], Marco Häberle[a], Daniel Merling[a], Steffen Lindner[a], Vladimir Gurevich[b], Florian Zeiger[c], Reinhard Frank[c], Michael Menth[a]

[a] *University of Tuebingen, Department of Computer Science, Chair of Communication Networks, Tuebingen, Germany*
[b] *Intel, Barefoot Division (BXD), United States of America*
[c] *Siemens AG, Corporate Technology, Munich, Germany*

Abstract

Programmable data planes allow users to define their own data plane algorithms for network devices including appropriate data plane application programming interfaces (APIs) which may be leveraged by user-defined software-defined networking (SDN) control. This offers great flexibility for network customization, be it for specialized, commercial appliances, e.g., in 5G or data center networks, or for rapid prototyping in industrial and academic research. Programming protocol-independent packet processors (P4) has emerged as the currently most widespread abstraction, programming language, and concept for data plane programming. It is developed and standardized by an open community, and it is supported by various software and hardware platforms.

In the first part of this paper we give a tutorial of data plane programming models, the P4 programming language, architectures, compilers, targets, and data plane APIs. We also consider research efforts to advance P4 technology. In the second part, we categorize a large body of literature of P4-based applied research into different research domains, summarize the contributions of these papers, and extract prototypes, target platforms, and source code availability. For each research domain, we analyze how the reviewed works benefit from P4's core features. Finally, we discuss potential next steps based on our findings.

Keywords: P4, SDN, programmable data planes

Email addresses: `frederik.hauser@uni-tuebingen.de` (Frederik Hauser), `marco.haeberle@uni-tuebingen.de` (Marco Häberle), `daniel.merling@uni-tuebingen.de` (Daniel Merling), `steffen.lindner@uni-tuebingen.de` (Steffen Lindner), `vladimir.gurevich@intel.com` (Vladimir Gurevich), `florian.zeiger@siemens.com` (Florian Zeiger), `reinhard.frank@siemens.com` (Reinhard Frank), `menth@uni-tuebingen.de` (Michael Menth)

1. Introduction

Traditional networking devices such as routers and switches process packets using data and control plane algorithms. Users can configure control plane features and protocols, e.g., via CLIs, web interfaces, or management APIs, but the underlying algorithms can be changed only by the vendor. This limitation has been broken up by SDN and even more by data plane programming.

SDN makes network devices programmable by introducing an API that allows users to bypass the built-in control plane algorithms and to replace them with self-defined algorithms. Those algorithms are expressed in software and typically run on an SDN controller with an overall view of the network. Thereby, complex control plane algorithms designed for distributed control can be replaced by simpler algorithms designed for centralized control. This is beneficial for use cases that are demanding with regard to flexibility, efficiency and security, e.g., massive data centers or 5G networks.

Programmable data planes enable users to implement their own data plane algorithms on forwarding devices. Users, e.g., programmers, practitioners, or operators, may define new protocol headers and forwarding behavior, which is without programmable data planes only possible for a vendor. They may also add data plane APIs for SDN control.

Data plane programming changes the power of the users as they can build custom network equipment without any compromise in performance, scalability, speed, or power on appropriate platforms. There are different data plane programming models, each with many implementations and programming languages. Examples are Click [1], VPP [2], NPL [3], and SDNet [4].

Programming protocol-independent packet processors (P4) is currently the most widespread abstraction, programming language, and concept for data plane programming. First published as a research paper in 2014 [5], it is now developed and standardized in the P4 Language Consortium, it is supported by various software- and hardware-based target platforms, and it is widely applied in academia and industry.

In the following, we clarify the contribution of this survey, point out its novelty, explain its organization, and provide a table with acronyms frequently used in this work.

1.1. Contributions

This survey pursues two objectives. First, it provides a comprehensive introduction and overview of P4. Second, it surveys publications describing applied research based on P4 technology. Its main contributions are the following:

- We explain the evolution of data plane programming with P4, relate it to prior developments such as SDN, and compare it to other data plane programming models.
- We give an overview of data plane programming with P4. It comprises the P4 programming language, architectures, compilers, targets, and data

plane APIs. These sections do not only include foundations but also present related work on advancements, extensions, or experiences.

- We summarize research efforts to advance P4 data planes. It comprises optimization of development and deployment, testing and debugging, research on P4 targets, and advances on control plane operation.

- We analyze a large body of literature considering P4-based applied research. We categorize 245 research papers into different application domains, summarize their key contributions, and characterize them with respect to prototypes, target platforms, and source code availability. For each research domain, we analyze how the reviewed works benefit from P4's core features.

We consider publications on P4 that were published until the end of 2020 and selected paper from 2021. Beside journal, conference, and workshop papers, we also include contents from standards, websites, and source code repositories. The paper comprises 519 references out of which 377 are scientific publications: 73 are from 2017 and before, 66 from 2018, 113 from 2019, 116 from 2020, and 9 from 2021.

1.2. Novelty

There are numerous surveys on SDN published in 2014 [6, 7], 2015 [8, 9, 10], and 2016 [11, 12] as well as surveys on OpenFlow (OF) from 2014 [13, 14, 15]. Only one of them [12] mentions P4 in a single sentence. Two surveys of data plane programming from 2015 [10, 9] were published shortly after the release of P4, one conference paper from 2018 [16] and a survey from 2019 [17] present P4 just as one among other data plane programming languages. Likewise, Michel et al. [18] gives an overview of data plane programming in general and P4 is one among other examined abstractions and programming languages. Our survey is dedicated to P4 only. It covers more details of P4 and a many more papers of P4-based applied research which have mostly emerged only within the last two years.

A recent survey focusing on P4 data plane programming has been published in [19]. The authors introduce data plane programming with P4, review 33 research works from four research domains, and discuss research issues. Another recent technical report [20] reviews 150 research papers from seven research domains. While typical research areas of P4 are covered, others (e.g., industrial networking, novel routing and forwarding schemes, and time-sensitive networking) are not part of the literature review. The different aspects of P4, e.g., the programming language, architectures, compilers, targets, data plane APIs, and their advancements are not treated in the paper. In contrast to both surveys on P4, we cover a greater level of detail of P4 technology and their advancements, and our literature review is more comprehensive.

1.3. Paper Organization

Figure 1 depicts the structure of this paper which is divided into two main parts: an *overview of P4* and a *survey of research publications*.

In the first part, Section 2 gives an introduction to network programmability. We describe the development from traditional networking and SDN to data plane programming and present the two most common data plane programming models. In Section 3, we give a technology-oriented tutorial of P4 based on its latest version $P4_{16}$. We introduce the P4 programming language and describe how user-provided P4 programs are compiled and executed on P4 targets. Section 4 presents the concept of P4 architectures as intermediate layer between the P4 programs and the targets. We introduce the four most common architectures in detail and describe P4 compilers. In Section 5, we categorize and present platforms that execute P4 programs, so-called P4 targets that are based on software, FPGAs, ASICs, or NPUs. Section 6 gives an introduction to data plane APIs. We describe their functions, present a characterization, introduce the four main P4 data plane APIs that serve as interfaces for SDN controllers, and point out controller use case patterns. In Section 7, we summarize research efforts that aim to improve P4 data plane programming.

The second part of the paper surveys P4-based applied research in communication networks. In Section 8, we classify core features of P4 that make it attractive for the implementation of data plane algorithms. We use these properties in later sections to effectively reason about P4's value for the implementation of various prototypes. We present an overview of the research domains and compile statistics about the included publications. The superordinate research domains are monitoring (Section 9), traffic management and congestion control (Section 10), routing and forwarding (Section 11), advanced networking (Section 12), network security (Section 13), and miscellaneous (Section 14) to cover additional, different topics. Each category includes a table to give a quick overview of the analyzed papers with regard to prototype implementations, target platforms, and source code availability. At the end of each section, we analyze how the reviewed works benefit from P4's core features.

In Section 15 we discuss insights from this survey and give an outlook on potential next steps. Section 16 concludes this work.

1.4. List of Acronyms

The following acronyms are used in this paper.

ACL	access control list
ALU	arithmetic logic unit
API	application programming interface
AQM	active queue management
ASIC	application-specific integrated circuit
AWW	adjusting advertised windows
bmv2	Behavioral Model version 2

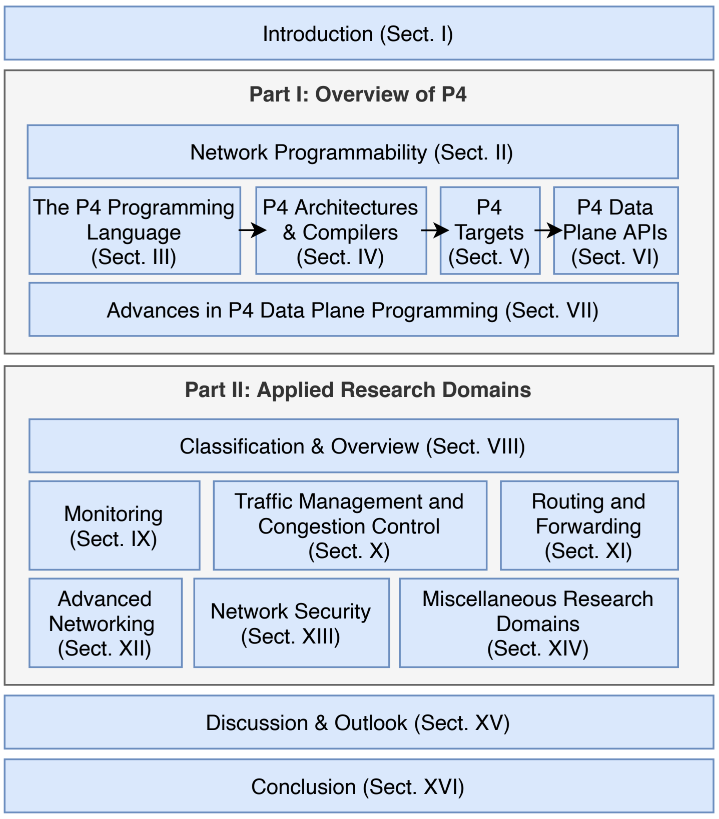

Introduction (Sect. I)
Part I: Overview of P4
Network Programmability (Sect. II)
The P4 Programming Language (Sect. III)
P4 Architectures & Compilers (Sect. IV)
P4 Targets (Sect. V)
P4 Data Plane APIs (Sect. VI)
Advances in P4 Data Plane Programming (Sect. VII)
Part II: Applied Research Domains
Classification & Overview (Sect. VIII)
Monitoring (Sect. IX)
Traffic Management and Congestion Control (Sect. X)
Routing and Forwarding (Sect. XI)
Advanced Networking (Sect. XII)
Network Security (Sect. XIII)
Miscellaneous Research Domains (Sect. XIV)
Discussion & Outlook (Sect. XV)
Conclusion (Sect. XVI)

Figure 1: Organization of the paper.

BGP	Border Gateway Protocol
BPF	Berkeley Packet Filter
CLI	command line interface
DAG	directed acyclic graph
DDoS	distributed denial of service
DPI	deep packet inspection
DPDK	Data Plane Development Kit
DSL	domain-specific language
eBPF	Extended Berkeley Packet Filter
ECN	Explicit Congestion Notification
FPGA	field programmable gate array
FSM	finite state machine
GTP	GPRS tunneling protocol

HDL	hardware description language
HLIR	high-level intermediate representation
IDE	integrated development environment
IDL	Intent Definition Language
IDS	intrusion detection system
INT	in-band network telemetry
LDWG	Language Design Working Group
LPM	longest prefix matching
LUT	look up table
MAT	match-action-table
ML	machine learning
NDN	named data networking
NF	network function
NFP	network flow processing
NFV	network function virtualization
NIC	network interface card
NPU	network processing unit
ODM	original design manufacturer
ODP	Open Data Plane
OEM	original equipment manufacturer
OF	OpenFlow
ONF	Open Networking Foundation
OVS	Open vSwitch
PISA	Protocol Independent Switching Architecture
PSA	Portable Switch Architecture
REG	register
RPC	remote procedure call
RTL	register-transfer level
SDK	software development kit
SDN	software-defined networking
SF	service function
SFC	service function chain
SRAM	static random-access memory
TCAM	ternary content-addressable memory
TSN	Time-Sensitive Networking

TNA	Tofino Native Architecture
uBPF	user-space BPF
VM	virtual machine
VNF	virtual network function
VPP	Vector Packet Processors
WG	working group
XDP	eXpress Data Path

2. Network Programmability

In this section, we first define the notion of network programmability and related terms. Then, we discuss control plane programmability and data plane programming, elaborate on data plane programming models, and point out the benefits of data plane programming.

2.1. Definition of Terms

We define *programmability* as the ability of the software or the hardware to execute an externally defined processing algorithm. This ability separates programmable entities from *flexible* (or *configurable*) ones; the latter only allow changing different parameters of the internally defined algorithm which stays the same.

Thus, the term *network programmability* means the ability to define the processing algorithm executed in a network and specifically in individual processing nodes, such as switches, routers, load balancers, etc. It is usually assumed that no special processing happens in the links connecting network nodes. If necessary, such processing can be described as if it takes place on the nodes that are the endpoints of the links or by adding a "bump-in-the-wire" node with one input and one output.

Traditionally, the algorithms, executed by telecommunication devices, are split into three distinct classes: the data plane, the control plane, and the management plane. Out of these three classes, the management plane algorithms have the smallest effect on both the overall packet processing and network behavior. Moreover, they have been programmable for decades, e.g., SNMPv1 was standardized in 1988 and created even earlier than that. Therefore, management plane algorithms will not be further discussed in this section.

True network programmability implies the ability to specify and change both the control plane and data plane algorithms. In practice this means the ability of network operators (users) to define both data and control plane algorithms on their own, without the need to involve the original designers of the network equipment. For the network equipment vendors (who typically design their own control plane anyway), network programmability mostly means the ability to define data plane algorithms without the need to involve the original designers of the chosen packet processing application-specific integrated circuit (ASIC).

Network programmability is a powerful concept that allows both the network equipment vendors and the users to build networks ideally suited to their needs. In addition, they can do it much faster and often cheaper than ever before and without compromising the performance or quality of the equipment.

For a variety of technical reasons, different layers became programmable at different point in time. While the management plane became programmable in the 1980s, control plane programmability was not achieved until late 2000s to early 2010s and a programmable switching ASICs did not appear till the end of 2015.

Thus, despite the focus on data plane programmability, we will start by discussing control plane programmability and its most well-known embodiment, called software-defined networking (SDN). This discussion will also better prepare us to understand the significance of data plane programmability.

2.2. Control Plane Programmability and SDN

Traditional networking devices such as routers or switches have complex data and control plane algorithms. They are built into them and generally cannot be replaced by the users. Thus, the functionality of a device is defined by its vendor who is the only one who can change it. In industry parlance, vendors are often called original equipment manufacturers (OEMs).

Software-defined networking (SDN) was historically the first attempt to make the devices, and *specifically their control plane*, programmable. On selected systems, device manufacturers allowed users to bypass built-in control plane algorithms so that the users can introduce their own. These algorithms could then directly supply the necessary forwarding information to the data plane which was still non-replaceable and remained under the control of the device vendor or their chosen silicon provider.

For a variety of technical reasons, it was decided to provide an APIs that could be called remotely and that is how SDN was born. Figure 2 depicts SDN in comparison to traditional networking. Not only the control plane became programmable, but it also became possible to implement network-wide control plane algorithms in a centralized controller. In several important use cases, such as tightly controlled, massive data centers, these centralized, network-wide algorithms proved to be a lot simpler and more efficient, than the traditional algorithms (e.g. Border Gateway Protocol (BGP)) designed for decentralized control of many autonomous networks.

The effort to standardize this approach resulted in the development of OpenFlow (OF) [21]. The hope was that once OF standardized the messaging API to control the data plane functionality, SDN applications will be able to leverage the functions offered by this API to implement network control. There is a huge body of literature giving an overview of OF [13, 14, 15] and SDN [6, 7, 8, 9, 11, 10, 12].

However, it soon became apparent that OF assumed a specific data plane functionality which was not formally specified. Moreover, the specific data plane, that served as the basis for OF, could not be changed. It executed the sole, although relatively flexible, algorithm defined by the OF specifications.

In part, it was this realization that led to the development of modern data plane programming that we discuss in the following section.

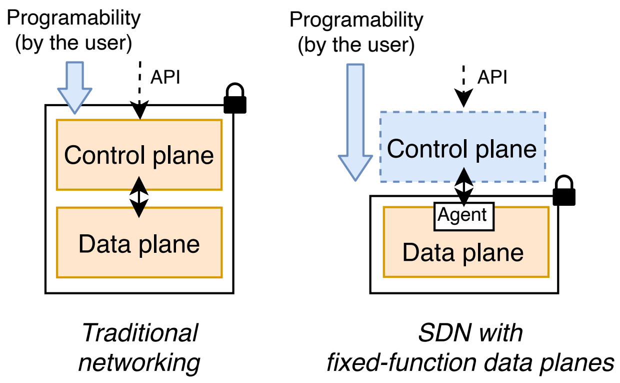

Programability (by the user)
API
Control plane
Data plane
Traditional networking
Programability (by the user)
API
Control plane
Agent
Data plane
SDN with fixed-function data planes

Figure 2: Distinction between traditional networking and SDN with fixed-function data planes.

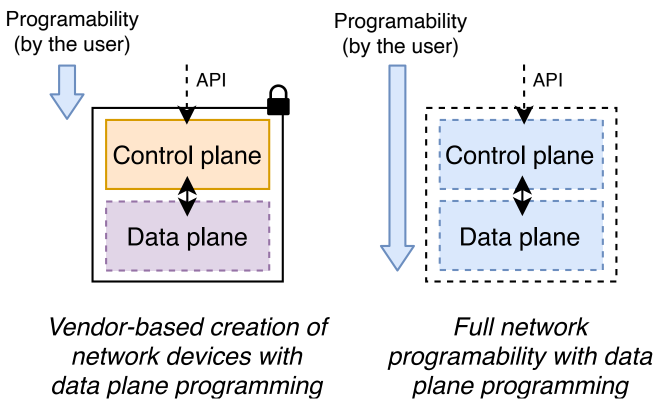
2.3. Data Plane Programming

As mentioned above, data plane programmability means that the data plane with its algorithms can be defined by the users, be they network operators or equipment designers working with a packet processing ASIC. In fact, data plane programmability existed during most of the networking industry history because data plane algorithms were typically executed on general-purpose CPUs. It is only with the advent of high-speed links, exceeding the CPU processing capabilities, and the subsequent introduction of packet processing (switching) ASICs that data plane programmability (or lack thereof) became an issue.

The data plane algorithms are responsible for processing all the packets that pass through a telecommunication system. Thus, they ultimately define the functionality, performance, and the scalability of such systems. Any attempt to implement data plane functionality in the control plane typically leads to significant performance degradation. When data plane programming is provided to users, it qualitatively changes their power. They can build custom network equipment without any compromise in performance, scalability, speed, or energy consumption.

For custom networks, new control planes and SDN applications can be designed and for them users can design data plane algorithms that fit them ideally. Data plane programming does not necessarily imply any provision of APIs for users nor does it require support for outside control planes as in OF. Device vendors might still decide to develop a proprietary control plane and use data plane programming only for their own benefit without necessarily making their systems more open (although many do open their systems now). Figure 3 visualizes both options.

Four surveys from [10, 9, 16, 17] give an overview on data plane programming, but do not set a particular focus to P4.



Programability (by the user)
API
Control plane
Data plane
Vendor-based creation of network devices with data plane programming
Programability (by the user)
API
Control plane
Data plane
Full network programability with data plane programming

Figure 3: Data plane programmability may be used by vendors for more efficient development or by users to provide own data and control plane algorithms.

2.4. Data Plane Programming Models

Data plane algorithms can and often are expressed using standard programming languages. However, they do not map very well onto specialized hardware such as high-speed ASICs. Therefore, several data plane models have been proposed as abstractions of the hardware. Data plane programming languages are tailored to those data plane models and provide ways to express algorithms for them in an abstract way. The resulting code is then compiled for execution on a specific packet processing node supporting the respective data plane programming model.

Data flow graph abstractions and the Protocol Independent Switching Architecture (PISA) are examples for data plane models. We give an overview of the first and elaborate in-depths on the second as PISA is the data plane programming model for P4.

2.4.1. Data Flow Graph Abstractions

In these data plane programming models, packet processing is described by a directed graph. The nodes of the graph represent simple, reusable primitives that can be applied to packets, e.g., packet header modifications. The directed edges of the graph represent packet traversals where traversal decisions are performed in nodes on a per-packet basis. Figure 4 shows an exemplary graph for IPv4 and IPv6 packet forwarding.

Examples for programming languages that implement this data plane programming model are Click [1], Vector Packet Processors (VPP) [2], and BESS [22].

2.4.2. Protocol-Independent Switching Architecture (PISA)

Figure 5 depicts the PISA. It is based on the concept of a programmable match-action pipeline that well matches modern switching hardware. It is a gen-

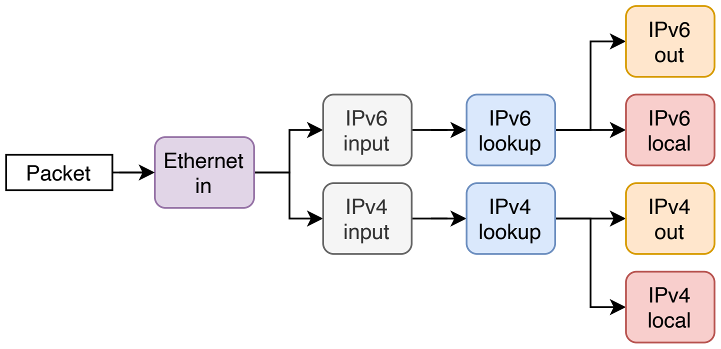


Figure 4: Data flow graph abstraction: example graph for IPv4 and IPv6 forwarding.

eralization of reconfigurable match-action tables (RMTs) [23] and disaggregated reconfigurable match-action tables (dRMTs) [24].

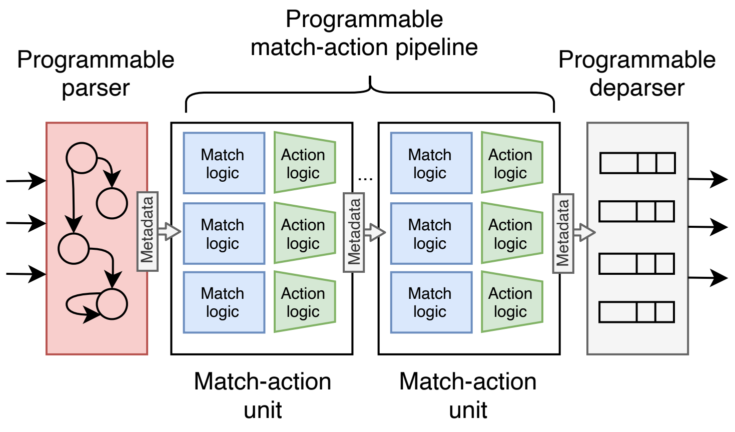


Figure 5: Protocol-Independent Switch Architecture (PISA).

PISA consists of a programmable parser, a programmable deparser, and a programmable match-action pipeline in between consisting of multiple stages.

- The *programmable parser* allows programmers to declare arbitrary headers together with a finite state machine that defines the order of the headers within packets. It converts the serialized packet headers into a well-structured form.

- The *programmable match-action pipeline* consists of multiple match-action units. Each unit includes one or more match-action-tables (MATs) to match packets and perform match-specific actions with supplied action data. The bulk of a packet processing algorithm is defined in the form of such MATs. Each MAT includes matching logic coupled with the memory (static random-access memory (SRAM) or ternary content-addressable memory (TCAM)) to store lookup keys and the corresponding action data. The action logic, e.g., arithmetic operations or header modifications, is implemented by arithmetic logic units (ALUs). Additional action logic can be implemented using stateful objects, e.g., counters, meters, or registers, that are stored in the SRAM. A control plane manages the matching logic by writing entries in the MATs to influence the runtime behavior.

- In the *programmable deparser*, programmers declare how packets are serialized.

A packet, processed by a PISA pipeline, consists of packet payload and packet metadata. PISA only processes packet metadata that travels from the parser all the way to the deparser but not the packet payload that travels separately.

Packet metadata can be divided into packet headers, user-defined and intrinsic metadata.

- *Packet headers* is metadata that corresponds to the network protocol headers. They are usually extracted in the parser, emitted in the deparser or both.
- *Intrinsic metadata* is metadata that relates to the fixed-function components. P4-programmable components may receive information from the fixed-function components by reading the intrinsic metadata they produce or control their behavior by setting the intrinsic metadata they consume.
- *User-defined metadata* (often referred as simply *metadata*) is a temporary storage, similar to local variables in other programming languages. It allows the developers to add information to packets that can be used throughout the processing pipeline.

All metadata, be it packet headers, user-defined or intrinsic metadata is *transient*, meaning that it is discarded when the corresponding packet leaves the processing pipeline (e.g., is sent out of an egress port or dropped).

PISA provides an abstract model that is applied in various ways to create concrete architectures. For example, it allows specifying pipelines containing different combinations of programmable components, e.g., a pipeline with no parser or deparser, a pipeline with two parsers and deparsers, and additional match-action pipelines between them. PISA also allows for specialized components that are required for advanced processing, e.g., hash/checksum calculations. Besides the programmable components of PISA, switch architectures typically also include configurable fixed-function components. Examples are ingress/egress port blocks that receive or send packets, packet replication engines that implements multicasting or cloning/mirroring of packets, and traffic managers, responsible for packet buffering, queuing, and scheduling.

The fixed-function components communicate with the programmable ones by generating and/or consuming intrinsic metadata. For example, the ingress port block generates ingress metadata that represents the ingress port number that might be used within the match-action units. To output a packet, the match-action units generates intrinsic metadata that represents an egress port number; this intrinsic metadata is then consumed by the traffic manager and/or egress port block.

Figure 6 depicts a typical switch architecture based on PISA. It comprises a programmable ingress and egress pipeline and three fixed-function components: an ingress block, an egress block, and a packet replication engine together with a traffic manager between ingress and egress pipeline.

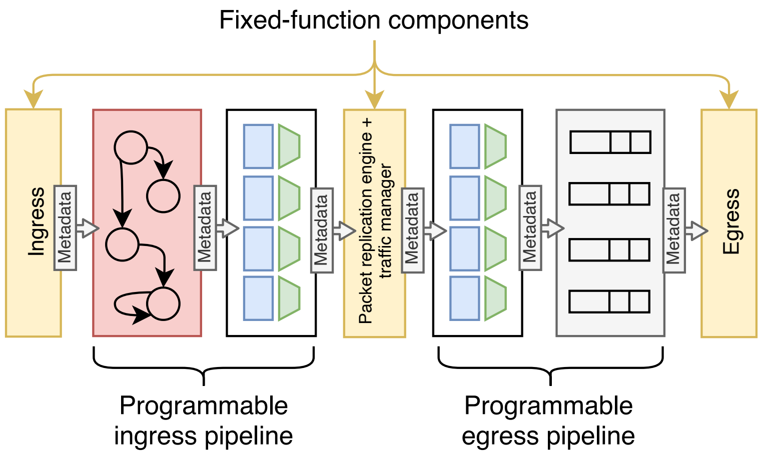

Fixed-function components
Ingress
Metadata
Metadata
Metadata
Packet replication engine + traffic manager
Metadata
Metadata
Metadata
Egress
Programmable ingress pipeline
Programmable egress pipeline

Figure 6: Exemplary switch architecture based on PISA.

P4 (Programming Protocol-Independent Packet Processors) [5] is the most widely used domain-specific programming language for describing data plane algorithms for PISA. Its initial idea and name were introduced in 2013 [25] and it was published as a research paper in 2014 [5]. Since then, P4 has been further developed and standardized by the P4 Language Consortium [26] that is part of the Open Networking Foundation (ONF) since 2019. The P4 Language Consortium is managed by a technical steering committee and hosts five working groups (WGs). $P4_{14}$ [27] was the first standardized version of the language. The current specification is $P4_{16}$ [28] which was first introduced in 2016.

Other data plane programming languages for PISA are FAST [29], Open-State [30], Domino [31], FlowBlaze [32], Protocol-Oblivious Forwarding [33], and NetKAT [34]. In addition, Broadcom [3] and Xilinx [4] offer vendor-specific programmable data planes based on match-action tables.

2.5. Benefits

Data plane programmability entails multiple benefits. In the following, we summarize key benefits.

Data plane programming introduces full flexibility to network packet processing, i.e., algorithms, protocols, features can be added, modified, or removed by the user. In addition, programmable data planes can be equipped with a user-defined API for control plane programmability and SDN. To keep complexity low, only components needed for a particular use case might be included in the code. This improves security and efficiency compared to multi-purpose appliances.

In conjunction with suitable hardware platforms, data plane programming allows network equipment designers and even users to experiment with new protocols and design unique applications; both do no longer depend on vendors of specialized packet-processing ASICs to implement custom algorithms. Compared to long development circles of new silicon-based solutions, new algorithms can be programmed and deployed in a matter of days.

Data plane programming is also beneficial for network equipment developers that can easily create differentiated products despite using the same packet processing ASIC. In addition, they can keep their know-how to themselves without the need to share the details with the ASIC vendor and potentially disclose it to their competitors that will use the same ASIC.

So far, modern data plane programs and programming languages have not yet achieved the degree of portability attained by the general-purpose programming languages. However, expressing data plane algorithms in a high-level language has the potential to make telecommunication systems significantly more target-independent. Also, data plane programming does not require but encourages full transparency. If the source code is shared, all definitions for protocols and behaviors can be viewed, analyzed, and reasoned about, so that data plane programs benefit from community development and review. As a result, users could choose cost-efficient hardware that is well suited for their purposes and run their algorithms on top of it. This trend has been fueled by SDN and is commonly known as network disaggregation.

3. The P4 Programming Language

We give an overview of the P4 programming language. We briefly recap its specification history and describe how P4 programs are deployed. We introduce the P4 processing pipeline and data types. We discuss parsers, match-action controls, and deparsers. Finally, we give an overview of tutorials and guides to P4.

3.1. Specification History

The P4 Language Design Working Group (LDWG) of the P4 Language Consortium has standardized so far two distinct standards of P4: $P4_{14}$ and $P4_{16}$. Table 1 depicts their specification history.

The $P4_{14}$ programming language dialect allows the programmers to describe data plane algorithms using a combination of familiar, general-purpose imperative constructs and more specialized declarative ones that provide support for

Table 1: Specification history of P4_{14} and P4_{16}.

P4_{14}	
Version 1.0.2	03/2015
Version 1.1.0	01/2016
Version 1.0.3	11/2016
Version 1.0.4	05/2017
Version 1.0.5	11/2018

P4_{16}	
Version 1.0.0	05/2017
Version 1.1.0	11/2018
Version 1.2.0	11/2018
Version 1.2.1	06/2020

the typical data-plane-specific functionality, e.g., counters, meters, checksum calculations, etc. As a result, the P4_{14} language core includes more than 70 keywords. It further assumed a specific pipeline architecture based on PISA.

P4_{16} has been introduced to address several P4_{14} limitations that became apparent in the course of its use. Those include the lack of means to describe various targets and architectures, weak typing and generally loose semantics (caused, in part, by the above-mentioned mix of imperative and declarative programming constructs), relatively low-level constructs, and weak support for program modularity.

Support for multiple different targets and pipeline architecture is the major contribution of the P4_{16} standard and is achieved by separating the core language from the specifics of a given architecture, thus making it architecture-agnostic. The structure, capabilities and interfaces of a specific pipeline are now encapsulated into an architecture description, while the architecture- or target-specific functions are accessible through an architecture library, typically provided by the target vendor. The core components are further structured into a small set of language constructs and a core library that is useful for most P4 programs. Compared to P4_{14}, P4_{16} introduced strict typing, expressions, nested data structures, several modularity mechanisms, and also removed declarative constructs, making it possible to better reason about the programs, written in the language. Figure 7 illustrates the concept which is subdivided into core components and architecture components.

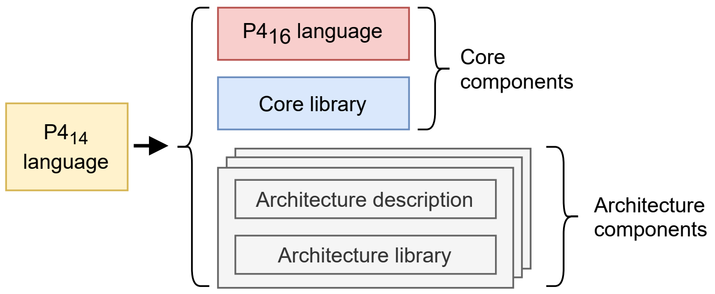


Figure 7: Comparison of the P4_{14} and P4_{16} language according to [28].

Due to the obvious advantages of P4_{16}, P4_{14} development has been discontinued, although it is still supported on a number of targets. Therefore, we focus on P4$_{16}$ in the remainder of this paper where P4 implicitly stands for P4$_{16}$.

3.2. Development and Deployment Process

Figure 8 illustrates the development and deployment process of P4 programs.

P4-programmable nodes, so-called P4 targets, are available as software or specialized hardware (see Section 5). They feature packet processing pipelines consisting of both P4-programmable and fixed-function components. The exact structure of these pipelines is target-specific and is described by a corresponding P4 architecture model (see Section 4) which is provided by the manufacturer of the target.

P4 programs are supplied by the user and are implemented for a particular P4 architecture model. They define algorithms that will be executed by the P4-programmable components and their interaction with the ones implemented in the fixed-function logic. The composition of the P4 programs and the fixed-function logic constitutes the full data plane algorithm.

P4 compilers (see Section 4) are also provided by the manufacturers. They translate P4 programs into target-specific code which is loaded and executed by the P4 target.

The P4 compiler also generates a data plane API that can be used by a user-supplied control plane (see Section 6) to manage the runtime behavior of the P4 target.

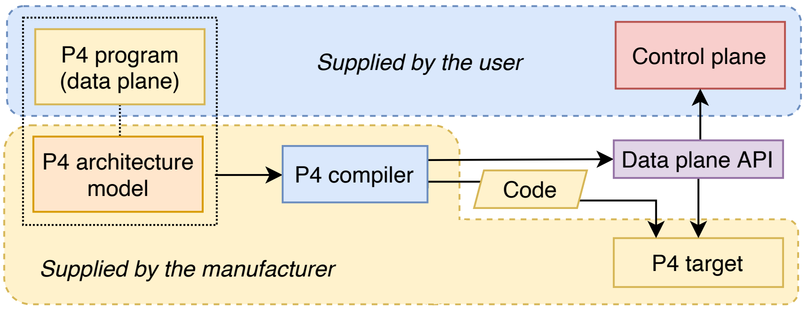


Figure 8: P4 deployment process according to [28].

3.3. Information Flow

P4$_{16}$ adopts PISA's concept of packet metadata. Figure 9 illustrates the information flow in the P4 processing pipeline. It comprises different blocks, where packet metadata (be it headers, user-defined or intrinsic metadata) is used to pass the information between them, therefore representing a uniform interface.

The parser splits up the received packet into individual headers and the remaining payload. Intrinsic metadata from the ingress block, e.g., the ingress port number or the ingress timestamp, is often provided by the hardware and can be made available for further processing. Many targets allow the user metadata to be initialized in the parser as well. Then, the headers and metadata are passed to the match-action pipeline that consists of one or more match-action units. The remaining payload travels separately and cannot be directly affected by the match-action pipeline processing.

While traversing the individual match-action pipeline units, the headers can be added, modified, or removed and additional metadata can be generated.

The deparser assembles the packet back by emitting the specified headers followed by the original packet payload. Packet output is configured with intrinsic metadata that includes information such as a drop flag, desired egress port, queue number, etc.

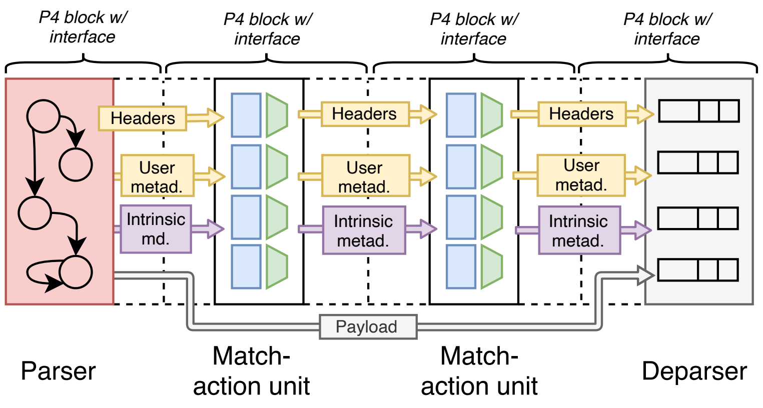


Figure 9: Information flow.

3.4. Data Types

P4$_{16}$ is a statically typed language that supports a rich set of data types for data plane programming.

3.4.1. Basic Data Types

P4$_{16}$ includes common basic types such as Boolean (`bool`), signed (`int`), and unsigned (`bit`) integers which are also known as bit strings. Unlike many common programming languages, the size of these integers is specified at *bit* granularity, with a wide range of supported widths. For example, types such as `bit<1>`, `int<3>`, `bit<128>` and wider are allowed.

In addition, P4 supports bit strings of variable width, represented by a special varbit type. For example, IPv4 options can be represented as `varbit<320>` since the size of IPv4 options ranges from zero to 10 32-bit words.

P4$_{16}$ also supports enumeration types that can be serializable (with the actual representation specified as `bit<N>` or `int<N>` during the type definition) or non-serializable, where the type representation is chosen by the compiler and hidden from the user.

3.4.2. Derived Data Types

Basic data types can be composed to construct derived data types. The most common derived data types are `header`, `header stack`, and `struct`.

The `header` data type facilitates the definition of packet protocol headers, e.g., IPv4 or TCP. A header consists of one more fields of the serializable types described above, typically `bit<N>`, serializable `enum`, or `varbit`. A header also has an implicit validity field indicating whether the header is part of a packet. The field is accessible through standard methods such as *setvalid()*, *setInvalid()*, and *isValid()*. Packet parsing starts with all headers being invalid. If the parser determines that a header is present in the packet, the header fields are extracted and the header's validity field is set valid. The standard packet *emit()* method used by a deparser equips packets only with valid headers. Thus, P4 programs can easily add and remove headers by manipulating their validity bits. A sample header declaration is shown in Figure 10.

A `header stack` is used to define repeating headers, e.g., VLAN tags or MPLS labels. It supports special operations allowing headers to be "pushed" onto the stack or "popped" from it.

`Struct` in P4 is a composed data type similar to structs in programming languages like C. Unlike the `header` data type, they can contain fields of any type including other structs, headers, and others.

```
typedef bit<48> macAddr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}
```

Figure 10: Sample declaration of the Ethernet header.

3.5. Parsers

Parsers extract header fields from ingress packets into header data and metadata. P4 does not include predefined packet formats, i.e., all required header formats including parsing mechanisms need to be part of the P4 program. Parsers are defined as finite state machine (FSM) with an explicit *Start* state, two ending states (*Accept* and *Reject*), and custom states in between.

Figure 11 depicts the structure of a typical P4 parser for Ethernet, MPLS, IPv4, TCP, and UDP headers. Figure 12 shows the source code fragment of the example parser in a $P4_{16}$ program. The process starts in the *Start* state and switches to the *Ethernet* state. In this state and the following states, information from the packet headers is extracted according to the defined header structure.

State transitions may be either conditional or unconditional. In the given example, the transition from the *Start* state to the *Ethernet* state is unconditional while in the *Ethernet* state the transition to the *MPLS*, *IPv4*, or *Reject*

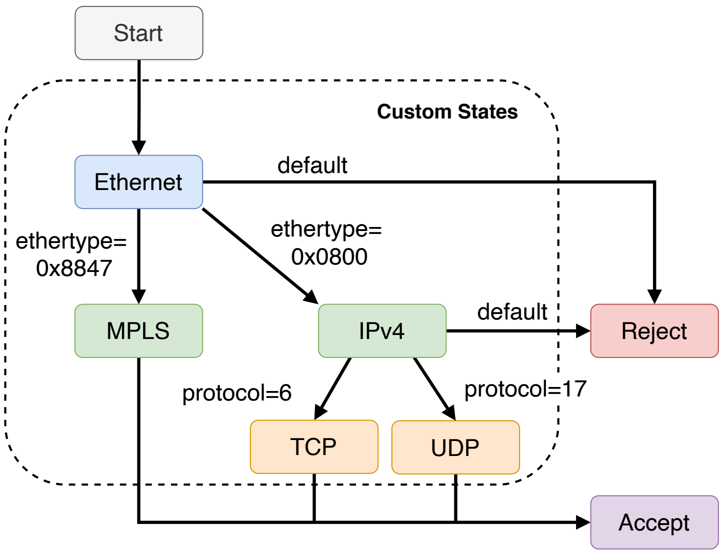


Figure 11: Example for the FSM of a P4 parser that parses packets with Ethernet, MPLS, IPv4, TCP, and UDP headers.

state depends on the value of the *EtherType* field of the extracted Ethernet header. Based on previously parsed header information, any number of further headers can be extracted from the packet. If the header order does not comply with the expected order, a packet can be discarded by switching to the *Reject* state. The parser can also implicitly transition into the *Reject* state in case of a parser exception, e.g., if a packet is too short.

3.6. Match-Action Controls

Match-action controls express the bulk of the packet processing algorithm and resemble traditional imperative programs. They are executed after successful parsing of a packet. In some architectures they are also called match-action pipeline units. In the following, we give an overview of control blocks, actions, and match-action tables.

3.6.1. Control Blocks

Control blocks, or just `controls`, are similar to functions in general-purpose languages. They are called by an *apply()* method. They have parameters and can call also other control blocks. The body of a control block contains the definition of resources, such as tables, actions, and externs that will be used for processing. Furthermore, a single *apply()* method is defined that expresses the processing algorithm.

P4 offers statements to express the program flow within a control block. Unlike common programming languages, P4 does not provide any statements that would allow the programmer to create loops. This ensures that all the

```
parser SampleParser(packet_in p, out headers h) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        p.extract(h.ethernet);
        transition select(h.ethernet.etherType) {
             0x8847: parse_mpls;
             0x0800: parse_ipv4;
            default: reject;
        };
    }

    state parse_ipv4 {
        p.extract(h.ipv4);
        transition select(h.ipv4.protocol) {
                  6: parse_tcp;
                 17: parse_udp;
            default: accept;
        }
    }

    state parse_udp {
        p.extract(h.udp);
        transition accept;
    }
    /* Other states follow */
}
```

Figure 12: Sample parser implementation of the FSM in Figure 11.

algorithms that can be coded in P4 can be expressed as directed acyclic graphs (DAGs) and thus are guaranteed to complete within a predictable time interval. Specific control statements include:

- a block statement `{}` that expresses sequential execution of instructions.
- an `if()` statement that expresses an execution predicated on a Boolean condition
- a `switch()` statement that expresses a choice from multiple alternatives
- an `exit()` statement that ends the control flow within a control block and passes the control to the end of the top-level control

Transformations are performed by several constructs, such as

- An assignment statement which evaluates the expression on its right-hand-side and assigns the result to a header or a metadata fields

- A match-action operation on a table expressed as the table's `apply()` method
- An invocation of an action or a function that encapsulate a sequence of statements
- An invocation of an extern method that represents special, target- and architecture-specific processing, often involving additional state, preserved between packets

A sample implementation of basic L2 forwarding is provided in Figure 13.

```
control SampleControl(inout headers h, inout standard_metadata_t
    standard_metadata) {

    action l2_forward(egressSpec_t port) {
        standard_metadata.egress_spec = port;
    }

    table l2 {
        key = {
            h.ethernet.dstAddr: exact;
        }
        actions = {
            l2_forward; drop;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if (h.ethernet.isValid()) {
            l2.apply();
        }
    }
}
```

Figure 13: Sample control block implementing basic L2 forwarding.

3.6.2. Actions

Actions are code fragments that can read and write packet headers and metadata. They work similarly to functions in other programming languages but have no return value. Actions are typically invoked from MATs. They can receive parameters that are supplied by the control plane as action data in MAT entries.

As in most general-purpose programming languages, the operations are written using expressions and the results are then assigned to the desired header or metadata fields. The operations available in P4 expressions include standard arithmetic and logical operations as well as more specialized ones such as bit slicing (`field[high:low]`), bit concatenation (`field1 ++ field2`), and saturated arithmetic (`|+|` and `|-|`).

Actions can also invoke methods of other objects, such as headers and architecture-specific externs, e.g., counters and meters. Other actions can also be called, similar to nested function calls in traditional programming languages.

Action code is executed sequentially, although many hardware targets support parallel execution. In this case, the compiler can optimize the action code for parallel execution as long as its effects are the same as in case of the sequential execution.

3.6.3. Match-Action Tables (MATs)

MATs are defined within control blocks and invoke actions depending on header and metadata fields of a packet. The structure of a MAT is declared in the P4 program and its table entries are populated by the control plane at runtime. A packet is processed by selecting a matching table entry and invoking the corresponding action with appropriate parameters.

The declaration of a MAT includes the match key, a list of possible actions, and additional attributes.

The match key consists of one or more header or metadata fields (variables), each with the assigned *match type*. The P4 core library defines three standard match types: exact, ternary, and longest prefix matching (LPM). P4 architectures may define additional match types, e.g., the *v1model* P4 architecture extends the set of standard match types with the range and selector match.

The list of possible actions includes the names of all actions that can be executed by the table. These actions can have additional, directional parameters which are provided as action data in table entries.

Additional attributes may include the size of the MAT, e.g., the maximum number of entries that can be stored in a table, a default action for a miss, or static table entries.

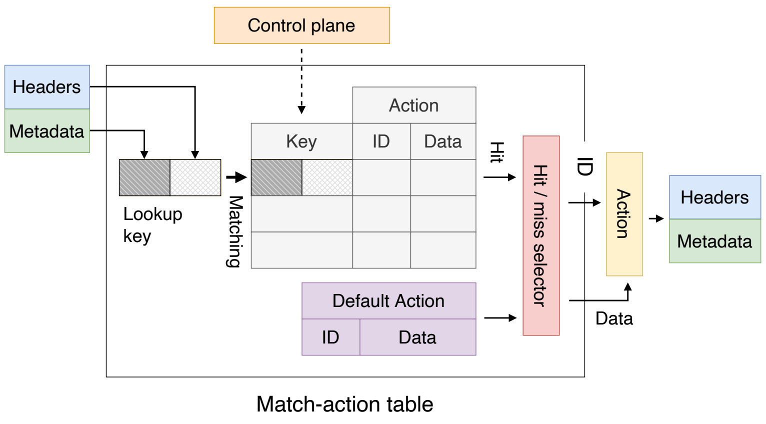


Figure 14: Structure of MATs in P4.

Figure 14 illustrates the principle of MAT operation. The MAT contains entries with values for match keys, the ID of the corresponding action to be invoked, and action data that serve as parameters for action invocation. For each packet, a lookup key is constructed from the set of header and metadata fields specified in the table definition. It is matched against all entries of the MAT using the rules associated with the individual field's match type. When the first match in the table is found, the corresponding action is called and the action data are passed to the action as directionless parameters. If no match is found in the table, a default action is applied.

As a special case, tables without a specified key always invoke the default action.

3.7. Deparser

The deparser is also defined as a control block. When packet processing by match-action control blocks is finished, the deparser serializes the packet. It reassembles the packet header and payload back into a byte stream so that the packet can be sent out via an egress port or stored in a buffer. Only valid headers are emitted, i.e., added to the packet. Thus, match-action control blocks can easily add and remove headers by manipulating their validity. Figure 15 provides a sample implementation.

```
control SampleDeparser(packet_out p, in headers h) {
    apply {
        p.emit(h.ethernet);
        p.emit(h.mpls);
        p.emit(h.ipv4);
        /* Normally, a packet can contain either
         * a TCP or a UDP header (or none at all),
         * but should never contain both
         */
        p.emit(h.tcp);
        p.emit(h.udp);
    }
}
```

Figure 15: Sample deparser implementation.

3.8. P4 Tutorials

The P4 Language Consortium provides a GitHub repository with simple programming exercises and a development VM containing all required software [35]. A guide on GitHub lists useful information for P4 newcomers, e.g. demo programs, information about other GitHub repositories, and an overview of P4 [36]. The Networked Systems Group at ETH Zürich provides resources for people who want to learn programming in P4, including lecture slides, references to useful documentation, examples and exercises [37].

4. P4 Architectures & Compilers

We present P4$_{16}$ architectures and introduce P4 compilers.

4.1. P4$_{16}$ Architectures

We summarize the concept of P4$_{16}$ architectures, describe externs, and give an overview of the most common P4$_{16}$ architectures.

4.1.1. Concept

As described before, P4$_{16}$ introduces the concept of P4 architectures as an intermediate layer between the core P4 language and the targets. A P4 architecture serves as programming models that represents the capabilities and the logical view of a target's P4 processing pipeline. P4 programs are developed for a specific P4 architecture. Such programs can be deployed on all targets that implement the same P4 architecture. The manufacturers of P4 targets provide P4 compilers that compile architecture-specific P4 programs into target-specific configuration binaries.

4.1.2. Externs

P4 architectures may provide additional functionalities that are not part of the P4 language core. Examples are checksum or hash computation units, random number generators, packet and byte counters, meters, registers, and many others. To make such extern functionalities usable, P4$_{16}$ introduces so-called *externs.*

Most of the externs have to be explicitly instantiated in P4 programs using their constructor method. The other methods provided by these externs can then be invoked on the given extern instance. Other externs (extern functions) do not require explicit instantiating.

Along with tables and value sets, P4 externs are allowed to preserve additional state between packets. That state may be accessible by the control plane, the data plane, or both. For example, the counter extern would preserve the number of packets or bytes that has been counted so that each new packet can properly increment it. The specifics of the state depend on the nature of the extern and cannot be specified in the language; this is done inside the vendor-specific API definitions.

While the P4 processing pipeline only allows packet header manipulation, extern functions may operate on packet payload as well.

4.1.3. Overview of Common P4$_{16}$ Architectures

We describe the four most common P4$_{16}$ architectures.

v1model. The v1model mimics the processing pipeline of P4_{14}. As depicted in Figure 16, it consists of a programmable parser, an ingress match action pipeline, a traffic manager, an egress match-action pipeline, and a deparser. It enables developers to convert P4_{14} programs into P4_{16} programs. Additional functionalities tracking the development of the reference P4 software switch Behavioral Model version 2 (bmv2) (see Section 5) are continuously added. All P4 examples in this paper are written using v1model.

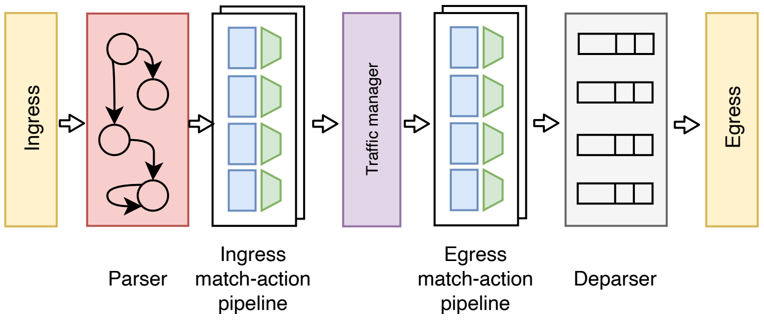


Figure 16: v1model architecture.

Portable Switch Architecture (PSA). The PSA is a P4 architecture created and further developed by the Architecture WG [38] in the P4 Language Consortium. Besides, the WG also discusses standard functionalities, APIs, and externs that every target mapping the PSA should support. Its last specification is Version 1.1 [39] from November 2018. Figure 17 illustrates the P4 processing pipeline of the PSA. It is divided into an ingress and egress pipeline. Each pipeline consists of the three programmable parts: parser, multiple control blocks, and deparser. The architecture also defines configurable fixed-function components.

PSA specifies several packet processing primitives, such as:

- Sending a packet to an unicast port
- Dropping a packet
- Sending the packet to a multicast group
- Resubmitting a packet, which moves the currently processed packet from the end of the ingress pipeline to the beginning of the ingress pipeline for the purpose of packet re-parsing
- Recirculating a packet, which moves the currently processed packet from the end of the egress pipeline to the beginning of the ingress pipeline for the purposes of recursive processing, e.g., tunneling
- Cloning a packet, which duplicates the currently processed packet. *Clone ingress to egress (CI2E)* creates a duplicate of the ingress packet at the end

of the ingress pipeline. *Clone egress to egress (CE2E)* creates a duplicate of the deparsed packet at the end of the egress pipeline. In both cases, cloned instances start processing at the beginning of the egress pipeline. Cloning can be helpful to implement powerful applications such as mirroring and telemetry.

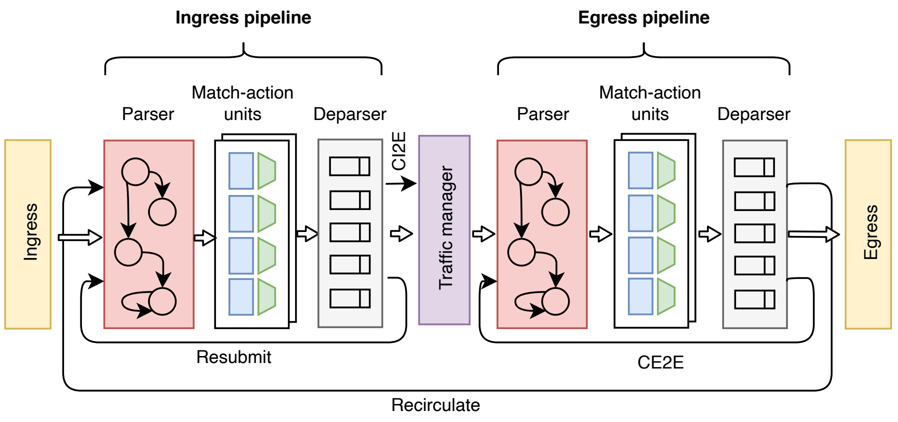


Figure 17: Portable Switch Architecture (PSA) with programmable and fixed-function parts and special packet processing primitives.

SimpleSumeArchitecture. The SimpleSumeArchitecture is a simplified P4 architecture that is implemented by FPGA-based P4 targets. As depicted in Figure 18, it features a parser, a programmable match-and-action pipeline, and a deparser.

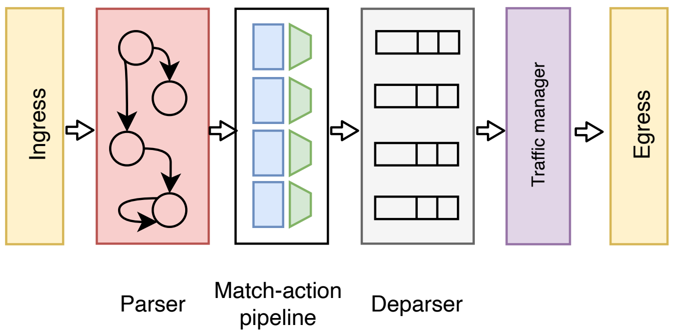


Figure 18: SimpleSumeArchitecture.

Tofino Native Architecture (TNA). TNA is a proprietary $P4_{16}$ architecture designed for Intel Tofino switching ASICs (see Section 5.3). Intel has published the architecture definitions and allows developers to publish programs written by using it.

The architecture describes a very high-performance, "industry-strength" device that is relatively complex. The basic programming unit is a so-called `Pipeline()` package that resembles an extended version of the Portable Switch Architecture (PSA) pipeline and consists of 6 top-level programmable components: the ingress parser, ingress match-action control, ingress deparser, and their egress counterparts. Since Tofino devices can have two or four processing pipelines, the final switch package can be formed anywhere from one to four distinct pipeline packages. More complex versions of the `Pipeline()` package allow the programmer to specify different parsers for different ports.

TNA also provides a richer set of externs compared to most other architectures. Most notable is TNA `RegisterAction()` which represents a small code fragment that can be executed on the register instead of simple read/write operations provided in other architectures. TNA provides a clear and consistent interface for mirroring and resubmit with additional metadata being passed via the packet byte stream. The same technique is also used to pass intrinsic metadata which greatly simplifies the design.

Additional externs that are not present in other architectures include low-pass filters, weighted random early discard externs, powerful hash externs that can compute CRC based on user-defined polynomials, ParserCounter, and others.

The set of intrinsic metadata in Tofino is also larger than in most other P4 architectures as presented before. Notable is support for two-level multicasting with additional source pruning, copy-to-cpu functionality, and support for IEEE 1588.

4.2. P4 Compiler

P4 compilers translate P4 programs into target-specific configuration binaries that can be executed on P4 targets. We first explain compilers based on the two-layer model which are most widely in use. Then we mention other compilers in less detail.

4.2.1. Two-Layer Compiler Model

Most P4 compilers use the two-layer model, consisting of a common frontend and a target-specific backend.

The frontend is common for all the targets and is responsible for parsing, syntactic and target-independent semantic analysis of the program. The program is finally transformed into an intermediate representation (IR) that is then consumed by the target-specific backend which performs target-specific transformations.

The first-generation P4 compiler for $P4_{14}$ was written in Python and used the so-called high-level intermediate representation (HLIR) [40] that represented $P4_{14}$ program as a tree of Python objects. The compiler is referred to as p4-hlir.

The new P4 compiler (p4c) [41] is written in C++ and uses C++-object-based IR. As an additional benefit, the IR can be output as a $P4_{16}$ program or a

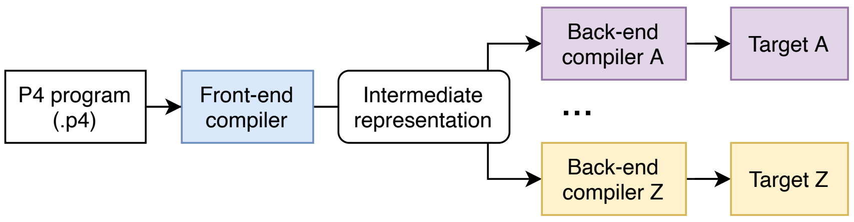


Figure 19: Structure and operation principle of P4 compilers using the two-layer model.

JSON file. The latter allows the developers and users to build powerful tools for program analysis without the need to augment the compiler. Figure 19 visualizes its structure and operating principle. The compiler consists of a generic frontend that accepts both P4_{14} and P4_{16} code which may be written for any architecture. It furthermore has several reference backends for the bmv2, eBPF, and uBPF P4 targets as well as a backend for testing purposes and a backend that can generate graphs of control flows of P4 programs. In addition, p4c provides the so-called "mid-end" which is a library of generic transformation passes that are used by the reference backends and can also be used by vendor-specific backends. The compiler is developed and maintained by P4.org.

P4 target vendors design and maintain their own compilers that include the common frontend. This ensures the uniformity of the language which is accepted by different compilers.

4.2.2. Other Compilers

MACSAD [42] is a compiler that translates P4 programs into Open Data Plane (ODP) [43] programs. Jose et al. [44] introduce a compiler that maps P4 programs to FlexPipe and RMT, two common software switch architectures. P4GPU [45] is a multistage framework that translates a P4 program into intermediate representations and other languages to eventually generate GPU code.

5. P4 Targets

We describe P4 targets based on software, FPGA, ASIC, and NPU. Table 2 compiles an overview of the targets, their supported architectures, and the current state of development.

5.1. Software-Based P4 Targets

Software-based P4 targets are packet forwarding programs that run on a standard CPU. We describe the 9 software-based P4 targets mentioned in Table 2.

Table 2: Overview of P4 targets.

Target	**P4 Version**	**$P4_{16}$ Architecture**	**Active Development**
Software			
p4c-behavioral	$P4_{14}$	n.a.	X
bmv2	$P4_{14}$, $P4_{16}$	v1model, psa	✓
eBPF	$P4_{16}$	ebpf_model.p4	✓
uBPF	$P4_{16}$	ubpf_model.p4	✓
XDP	$P4_{16}$	xdp_model.p4	✓
T4P4S	$P4_{14}$, $P4_{16}$	v1model, psa	✓
Ripple	n.a	n.a	n.a
PISCES	$P4_{14}$	n.a.	X
PVPP	n.a.	n.a.	X
ZodiacFX	$P4_{16}$	zodiacfx_model.p4	n.a.
FPGA			
P4→NetFPGA	$P4_{16}$	SimpleSumeSwitch	✓
Netcope P4	n.a.	n.a.	✓
P4FPGA	$P4_{14}$, $P4_{16}$	n.a.	X
ASIC			
Barefoot Tofino/Tofino 2	$P4_{14}$, $P4_{16}$	v1model, psa, TNA	✓
Pensando Capri	$P4_{16}$	n.a	✓
NPU			
Netronome	$P4_{14}$, $P4_{16}$	v1model	✓

5.1.1. *p4c-behavioural*

p4c-behavioral [46] is a combined P4 compiler and P4 software target. It was introduced with the first public release of P4. p4c-behavioral translates the given $P4_{14}$ program into an executable C program.

5.1.2. *Behavioral Model version 2 (bmv2)*

The second version of the P4 software switch Behavioral Model (bmv2) [47] was introduced to address the limitations of p4c-behavioural (see also [48]). In contrast to p4c-behavioral, the source code of bmv2 is static and independent of P4 programs. P4 programs are compiled to a JSON representation that is loaded onto the bmv2 during runtime. External functions and other extensions can be added by extending bmv2's C++ source code. bmv2 is not a single target, but a collection of targets [49]:

- *simple_switch* is the bmv2 target with the largest range of features. It contains all features from the $P4_{14}$ specification and supports the v1model architecture of $P4_{16}$. simple_switch includes a program-independent Thrift API for runtime control.

- *simple_switch_grpc* extends simple_switch by the P4Runtime API that is based on gRPC (see Section 6.3.1).
- *psa_switch* is similar to simple_switch, but supports PSA instead of v1model.
- *simple_router* and *l2_switch* support only parts of the standard metadata and do not support $P4_{16}$. They are intended to show how different architectures can be implemented with bmv2.

Although bmv2 is intended for testing purposes only, throughput rates up to 1 Gbit/s for a P4 program with IPv4 LPM routing have been reported [50]. bmv2 is under active development, i.e., new functionality is added frequently.

5.1.3. BPF-based Targets

Berkeley Packet Filters (BPFs) add an interface on a UNIX system that allows sending and receiving raw packets via the data link layer. User space programs may rely on BPFs to filter packets that are sent to it. BPF-based P4 targets are mostly intended for programming packet filters or basic forwarding in P4.

eBPF. Extended Berkeley Packet Filters (eBPFs) are an extension of BPFs for the Linux kernel. eBPF programs are dynamically loaded into the Linux kernel and executed in a virtual machine (VM). They can be linked to functions in the kernel, inserted into the network data path via iproute2, or bound to sockets or network interfaces. eBPF programs are always verified by the kernel before execution, e.g., programs with loops or backward pointers would not be executed. Due to their execution in a VM, eBPF programs can only access certain regions in memory besides the local stack. Accessing kernel resources is protected by a white list. eBPF programs may not block and sleep, and usage of locks is limited to prevent deadlocks. The p4c compiler features the *p4c-ebpf* back-end to compile $P4_{16}$ programs to eBPF [51].

uBPF. user-space BPFs (uBPFs) relocate the eBPF VM from the kernel space to the user space. *p4c-ubpf* [52] is a backend for p4c that compiles P4 HLIR for uBPF. In contrast to p4c-ebpf, it also supports packet modification, checksum calculation, and registers, but no counters.

XDP. eXpress Data Path (XDP) is based on eBPF and allows to load an eBPF program into the RX queue of a device driver. p4c-xdp [53] is a backend for p4c that compiles P4 HLIR for XDP. Similar to p4c-ubpf, it supports packet modification and checksum calculation. In contrast to p4c-ebpf, it supports counters instead of registers.

5.1.4. T_4P_4S

T_4P_4S (pronounced "tapas") [54, 55] is a software P4 target that relies on interfaces for accelerated packet processing such as Data Plane Development Kit (DPDK) [56] or Open Data Plane (ODP) [43]. T_4P_4S provides a compiler that translates P4 programs into target-independent C code that interfaces a network hardware abstraction library. Hardware-dependent and hardware-independent functionalities are separated from each other. Its source code is available on GitHub [57]. Bhardwaj et al. [58] describe optimizations for improving T_4P_4S performance by up to 15%.

5.1.5. Ripple

Ripple [59] is a P4 target based on DPDK. It uses a static universal binary that is independent of the P4 program. The data plane of the static binary is configured at runtime based on P4 HLIR. This results in a shorter downtime when updating a P4 program in contrast to targets like T_4P_4S. Ripple uses vectorization to increase the performance of packet processing.

5.1.6. PISCES

PISCES [60] transforms the Open vSwitch (OVS) [61] into a software P4 target. OVS is a popular SDN software switch that is designed for high throughput on virtualization platforms for flexible networking between VMs. The PISCES compiler translates P4 programs into C code that replace parts of the source code of OVS. This makes OVS dependent on the P4 program, i.e., OVS must be recompiled with every modification of the P4 program. PISCES does not support stateful components such as registers, counters, or meters. The developers claim that PISCES does not add performance overhead to OVS. As the last commit in the public repository [62] is from 2016, PISCES seems not to be under active development.

5.1.7. PVPP

PVPP [63, 64] integrates P4 programs into plugins for Vector Packet Processors (VPP) (see Section 2.4.1). The P4-to-PVPP compiler comprises two stages. First, a modified p4c compiler translates P4 programs into target-dependent JSON code. Then, a Python compiler translates the JSON code into a VPP plugin in C source code. According to the authors, performance decreases by 5-17% compared to VPP but is still significantly better than OVS. Unfortunately, the source code and further information are not available for the public.

5.1.8. ZodiacFX

The ZodiacFX is a lightweight development and experimentation board originally designed as OF switch featuring four Fast Ethernet ports. It is based on an Atmel processor and an Ethernet switching chip [65]. The authors provided an extension [66, 67] to run P4 programs on the board. P4 programs are compiled using an extended version of p4c and the p4c-zodiacfx backend compiler. Then, the result of this compilation is used to generate a firmware image. Zanna et al. [68] compare the performance of P4 and OF on that target, and find out that differences among all test cases are small.

5.2. FPGA-Based P4 Targets

Several tool chains translate P4 programs into implementations for field programmable gate arrays (FPGAs). The process includes logic synthesis, verification, validation, and placement/routing of the logic circuit for the FPGA. We describe the P4→NetFPGA, Netcope P4, and P4FPGA tool chain. Finally, we mention research results for FPGA-based P4 targets.

5.2.1. P4→NetFPGA

The P4→NetFPGA workflow [69, 70] provides a development environment for compiling and running P4 programs on the NetFPGA SUME board that provides four SFP+ ports [71]. The development environment is built around the P4-SDnet compiler and the SDnet data plane builder from Xilinx, i.e., a full license for the Xilinx Vivado design suite is needed. Custom external functions can be implemented in a hardware description language (HDL) such as Verilog and included in the final FPGA program. This also allows external IP cores to be integrated as P4 externs in P4 programs. The P4→NetFPGA tool chain supports P4$_{16}$ based on the P4 architecture SimpleSumeSwitch (see Section 4.1).

5.2.2. Netcope P4

Netcope P4 [72] is a commercial cloud service that creates FPGA firmware from P4 programs. Knowledge of HDL development is not needed and all necessary IP cores are provided by Netcope. The cloud service can be used in conjunction with the Netcope software development kit (SDK). This combination allows developers to combine the VHDL code of the cloud service with custom HDL code, e.g., from an external function. As target platform, Netcope P4 supports FPGA boards from Netcope, Silicom, and Intel that are based on Xilinx or Intel FPGAs.

5.2.3. P4FPGA

P4FPGA [73] is a P4$_{14}$ and P4$_{16}$ compiler and runtime for the Bluespec programming language that can generate code for Xilinx and Altera FPGAs. The last commit in the archived public repository [74] is from 2017.

5.2.4. Research Results

Benácek and Kubátová [75, 76] present how P4 parse graph descriptions can be converted to optimized VHDL code for FPGAs. The authors demonstrate how a complex parser for several header fields achieves a throughput of 100 Gbit/s on a Xilinx Virtex-7 FPGA while using 2.78% slice look up tables (LUTs) and 0.76% slice registers (REGs). In a follow-up work [77], the optimized parser architecture supports a throughput of 1 Tbit/s on Xilinx UltraScale+ FPGAs and 800 Gbit/s on Xilinx Virtex-7 FPGAs. Da Silva et al. [78] also investigate the high-level synthesis of packet parsers in FPGAs. Kekely and Korenek [79] describe how MATs can be mapped to FPGAs. Iša et al. [80] describe a system for automated verification of register-transfer level (RTL) generated from P4 source code. Cao et al. [81, 82] propose a template-based process to convert P4 programs to VHDL. They use a standard P4 frontend compiler to compile the P4 program into an intermediate representation. From this representation, a custom compiler maps the different elements of the P4 program to VHDL templates which are used to generate the FPGA code.

5.3. ASIC-Based P4 Targets

5.3.1. Intel Tofino

Intel Tofino is the world's first user programmable Ethernet switch ASIC. It is designed for very high throughput of 6.5 Tbit/s (4.88 B pps) with 65 ports running at 100 Gbit/s. Its successor, the Tofino 2 ASIC, supports throughput rates of up to 12.8 Tbit/s with ports running at up to 400 Gbit/s. Tofino has been built by Barefoot Networks, a former startup company that was acquired by Intel in 2019.

The Tofino ASIC implements the TNA, a custom P4 architecture that significantly extends PSA (see Section 4.1). It provides support for advanced device capabilities which are required to implement complex, industrial-strength data plane programs. The device comes with 2 or 4 independent packet processing pipelines (pipes), each capable of serving 16 100 Gbit/s ports. All pipes can run the same P4 program or each pipe can run its own program independently. Pipes can also be connected together, allowing the programmers to build programs requiring longer processing pipelines.

The Tofino ASIC processes packets at line rate irrespective of the complexity of the executed P4 program. This is achieved by a high degree of pipelining (each pipe is capable of processing hundreds of packets simultaneously) and parallelization. In addition to standard arithmetic and logical operations, Tofino provides specialized capabilities, often required by data plane programs, such as hash computation units and random number generators. For stateful processing Tofino offers counters, meters, and registers, as well as more specialized processing units. Some of them support specialized operations, such as approximate non-linear computations required to implement state-of-the-art data plane algorithms. Built-in packet generators allow the data plane designers to implement protocols, such as BFD, without using externally running control plane processes. These and other components are exposed through TNA which is openly published by Intel [83].

Tofino fixed-function components offer plenty of advanced functionality. The buffering engine has a unified 22 MB buffer, shared by all the pipes, that can be subdivided into several pools. Tofino Traffic Manager supports both store-and-forward as well as the cut-through mode, up to 32 queues per port, precise traffic shaping and multiple scheduling disciplines. Tofino provides nanosecond-precision timestamping that facilitates both the implementation of time synchronization protocols, such as IEEE 1588, as well as precise delay measurements. Additional intrinsic metadata support a variety of telemetry applications, such as INT.

The development is conducted using Intel P4 Studio which is a software development environment containing the P4 compiler, the driver, and other software necessary to program and manage the Tofino. A special interactive visualization tool (P4i) allows the developers to see the P4 program being mapped onto the specific hardware resources further assisting them in fitting and optimizing their programs. Intel P4 compiler for Tofino has special capabilities, allowing it to parallelize the code thereby taking advantage of the highly parallel nature of Tofino hardware.

A number of original design manufacturers (ODMs) produce open systems (white boxes) with the Tofino ASIC that are used for research, development, and production of custom systems. Examples include the EdgeCore Wedge 100BF-32X [84], APS Networks BF2556-1T-A1F [85] and BF6064-T-A2F [86], NetBerg Aurora 610 [87], and others.

Most white box systems follow a modern, server-like design with a separate board management controller, responsible for handling power supplies, fans, LEDs, etc., and a main CPU, typically x86_64, running a Linux operating system. The main CPU is connected to the Tofino ASIC via a PCIe interface. Some boards also provide one or more high-speed on-board Ethernet connections for faster packet interface. External Ethernet ports support speeds from 10 Gbit/s to 100 Gbit/s using standard QSFP28 cages although some systems offer lower-speed (1 Gbit/s) ports as well. Most of these systems are also powerful enough to support running development tools natively, e.g., a P4 compiler, even though this is not necessarily required.

Tofino ASICs are also used in proprietary network switches, e.g., by Arista [88] and Cisco [89]. Some Tofino-based switches are supported by Microsoft SONiC [90].

5.3.2. Pensando Capri

The Capri P4 Programmable Processor [91, 92] is an ASIC that powers network interface cards (NICs) by Pensando Systems aimed for cloud providers. It is coupled with fixed function components for cryptography operations like AES or compression algorithms and features multiple ARM cores.

5.4. NPU-Based P4 Targets

Network processing units (NPUs) are software-programmable ASICs that are optimized for networking applications. They are part of standalone network devices or device boards, e.g., PCI cards.

Netronome network flow processing (NFP) silicons can be programmed with P4 [93] or C [94]. A C-based programming model is available that supports program functions to access payloads and allows developing P4 externs. The Agilio P4C SDK consists of a tool chain including a backend compiler, host software, and a full-featured integrated development environment (IDE). All current Agilio SmartNICs based on NFP-4000, NFP-5000, and NFP-6480 are supported. Harkous et al. [95] investigate the impact of basic P4 constructs on packet latency on Agilio SmartNICs.

6. P4 Data Plane APIs

We introduce data plane APIs for P4, present a characterization, describe the three most commonly used P4 data plane APIs, and compare different control plane use cases.

6.1. Definition & Functionality

Control planes manage the runtime behavior of P4 targets via data plane APIs. Alternative terms are *control plane APIs* and *runtime APIs*. The data plane API is provided by a device driver or an equivalent software component. It exposes data plane features to the control plane in a well-defined way. Figure 20 shows the main control plane operations. Most important, data plane APIs facilitate runtime control of P4 entities (MATs and externs). They typically also comprise a packet I/O mechanism to stream packets to/from the control plane. They also include reconfiguration mechanisms to load P4 programs onto the P4 target. Control planes can control data planes only through data plane APIs, i.e., if a data plane feature is not exposed via a corresponding API, it cannot be used by the control plane.

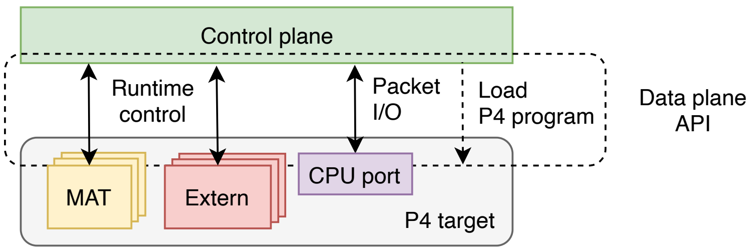


Figure 20: Runtime management of a P4 target by the control plane through the data plane API. The figure depicts the four most central operations: Runtime control of MATs and extern objects, packet-in/out, and loading of P4 programs.

It is important to note that P4 does not require a data plane APIs. P4 targets may also be used as a packet processor with a fixed behavior that is defined by the P4 program where static MAT entries are part of the P4 program itself.

6.2. Characterization of Data Plane APIs

Data plane APIs in P4 can be characterized by their level of abstraction, their dependency on the P4 program, and the location of the control plane.

6.2.1. Level of Abstraction

Data plane APIs can be characterized by their level of abstraction.

- *Device access APIs* provide direct access to hardware functionalities like device registers or memories. They typically use low-level mechanisms like DMA transactions. While this results in very low overhead, this type of API can be neither vendor- nor device-independent.

- *Data plane specific APIs* are APIs with a higher level of abstraction. They provide access to objects defined by the P4 program instead of hardware-specific parts. In contrast to device access APIs, vendor- and device-independence is possible for this type of API.

6.2.2. Dependency on the P4 Program

Data plane APIs can be characterized by their dependency on the P4 program.

- *Program-dependent APIs* have a set of functions, data structures, and other names that are derived from the P4 program itself. Therefore, they depend on the P4 program and are applicable to this P4 program only. If the corresponding P4 program is changed, function names, data structures, etc., might change, which requires a recompilation or modification of the control plane program.
- *Program-independent APIs* consist of a fixed set of functions that receives a list of P4 objects that are defined in the P4 program. Thus, the names of the API functions, data structures, etc., do not depend on the program and are universally applicable. If the corresponding P4 program changes, neither the names, nor the definitions of the API functions will change as long as the control plane "knows" the names of the right tables, fields and other object that need to be operated on. Program-independent APIs model configurable objects either with the *object-based* or the *table-based* approach. As known from object-oriented programming, the object-based approach relies on methods that are defined for each class of data plane objects. In contrast, the table-based approach treats every class of data plane object as a variation of a table. This reduces the number of API methods as only table manipulations need to be provided as methods.

6.2.3. Control Plane Location

Data plane APIs can be characterized by the location of the control plane.

- *APIs for local control* are implemented by the device driver and are executed on the local CPU of the device that hosts the programmable data plane. Usually, the APIs are presented as set of C function calls just like for other devices that operating system are accessing.
- *APIs for remote control* add the ability to invoke API calls from a separate system. This increases system stability and modularity, and is essential for SDN and other systems with centralized control. Remote control APIs follow the base methodology of remote procedure calls (RPCs) but rely on modern message-based frameworks that allow asynchronous communication and concurrent calls to the API. Examples are Thrift [96] or gRPC [97]. For example, gRPC uses HTTP/2 for transport and includes many functionalities ranging from access authentication, streaming, and flow control. The protocol's data structures, services, and serialization schemes are described with protocol buffers (protobuf) [98].

6.3. Data Plane API Implementations

We introduce the three most common data plane APIs: P4Runtime, Barefoot Runtime Interface (BRI), and BM Runtime. All of them are data-plane specific and program-independent. Table 3 lists their properties that have been introduced before.

6.3.1. P4Runtime API

P4Runtime is one of the most commonly used data plane APIs that is standardized in the API WG [99] of the P4 Language Consortium. For implementing the RPC mechanisms, it relies on the gRPC framework with protobuf. Its most recent specification v1.3.0 [100] was published in December 2020.

Operating Principle. Figure 21 depicts the operating principle of P4Runtime. P4 targets include a gRPC server, controllers implement a gRPC client. To protect the gRPC connection, TLS with optional mutual certificate authentication can be enabled. The API structure of P4Runtime is described within the `p4runtime.proto` definition. The gRPC server on P4 targets interacts with the P4-programmable components via platform drivers. It has access to P4 entities (MATs or externs) and can load target-specific configuration binaries. The structure of the API calls to access P4 entities are described in the `p4info.proto`. It is part of the P4Runtime but developers can extend it to use custom data structures, e.g., to implement interaction with target-specific externs. P4Runtime provides support for multiple controllers. For every P4 entity, read access is provided to all controllers whereas write access is only provided to one controller. To manage this access, P4 entities can be arranged in groups where each group is assigned to one primary controller with write access and arbitrary, secondary controllers with read access. Interaction between controllers and P4 targets works as follows. P4 compilers (see Section 4.2) with support for P4Runtime generate a P4Runtime configuration. It consists of the target-specific configuration binaries and P4Info metadata. P4Info describes all P4 entities (MATs and externs) that can be accessed by controllers via P4Runtime. Then, the controllers establish a gRPC connection to the gRPC server on the P4 target. The target-specific configuration is loaded onto the P4 target and P4 entities can be accessed.

Implementations. gRPC and protobuf libraries are available for many high-level programming languages such as C++, Java, Go, or Python. Thereby, P4Runtime can be implemented easily on both controllers and P4 targets.

- *Controllers*: P4Runtime is supported by most common SDN controllers. P4 brigade [101] introduces support for P4Runtime on the Open Network Operating System (ONOS). OpenDaylight (ODL) introduces support for P4Runtime via a plugin [102]. Stratum [103] is an open-source network operating system that includes an implementation of the P4Runtime and OpenConfig interfaces. Custom controllers, e.g., for P4 prototypes, can be implemented in Python with the help of the p4runtime_lib [104].

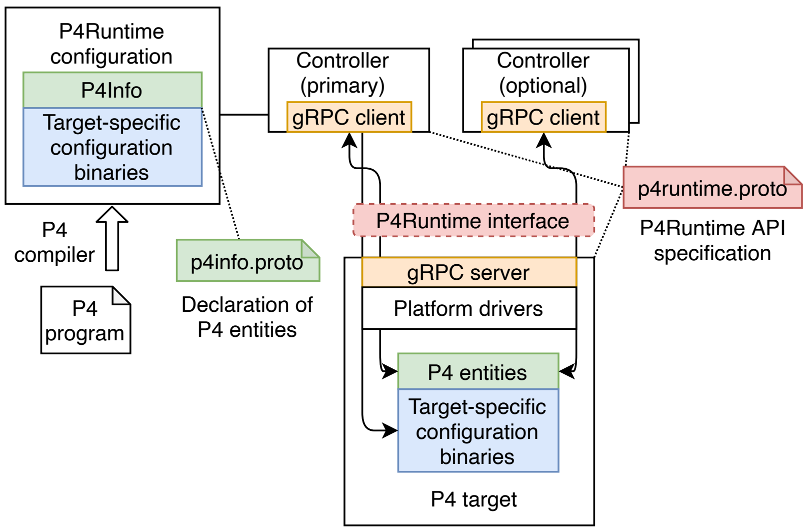


Figure 21: P4Runtime architecture (similar to [100]).

- *Targets*: The *PI Library* [105] is the open-source reference implementation of a P4Runtime gRPC server in C. It implements functionality for accessing MATs and supports extensions for target-specific configuration objects, e.g., registers of a hardware P4 target. The PI Library is used by many P4 targets including bmv2 [106] and the Tofino.

6.3.2. Barefoot Runtime Interface (BRI)

The BRI consists of two independent APIs that are available on Tofino-based P4 hardware targets. The *BfRt API* is an API for local control. It includes C, C++ and Python bindings that can be used to implement control plane programs. The *BF Runtime* is an API for remote control. As for P4Runtime, it is based on the gRPC RPC framework and protobuf, i.e., bindings for different languages are available. An additional Python library implements a simpler, BfRt-like interface for cases where simplicity is more essential than the performance of BF Runtime.

6.3.3. BM Runtime API

BM Runtime API is a program-independent data plane API for the bmv2 software target. It relies on the Thrift RPC framework. bmv2 includes a command line interface (CLI) program [107] to manipulate MATs and configure the multicast engine of the bmv2 P4 software target via this API.

6.4. Controller Use Case Patterns

We present three use case patterns which are abstractions of the controller use cases introduced in the P4Runtime specification [100]. However, these are

Table 3: Characterization of data plane specific APIs.

API	**Program independence**	**Control plane location**
P4Runtime	✓	Remote (gRPC)
BF Runtime	✓	Remote (gRPC)
BfRt API	✓	Local (C, C++ and Python bindings)
BM Runtime	✓	Remote (Thrift RPC)

neither conclusive nor complete as derivations or extensions are possible.

6.4.1. Embedded/Local Controller

P4 hardware targets (see Section 5) comprise or are attached to a computing platform. This facilitates running controllers directly on the P4 target. Figure 22 depicts this setup. The controller application may either use a local API, e.g., C calls, or just execute a controller application that interfaces the data plane via an RPC channel.

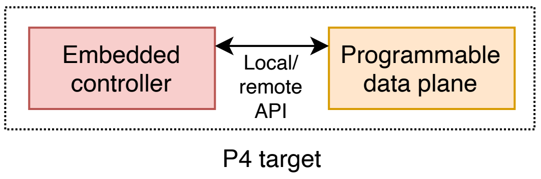


Figure 22: Embedded/local controller use case pattern. The P4 target comprises an embedded controller that is running a control plane program.

6.4.2. Remote Controllers

Remote controllers resemble the typical SDN setup where data plane devices are managed by a centralized control plane with an overall view on the network. Controllers need to be protected against outages and capacity overload, i.e., they need to be replicated for fail-safety and scalability. Figure 23 depicts two possible use cases. In the first shown use case (a), the programmable data plane on the P4 target is managed by remote controllers. In the second shown use case (b), the P4 target is managed by both, the embedded controller and remote controllers. Remote controllers might be interfaced using the remote API of the programmable data plane or an arbitrary API that is provided by the embedded controller. This option is often used for the implementation of so-called *hierarchical control plane* structures where control plane functionality is distributed among different layers. Control plane functions that do not require a global view of the network, e.g., link discovery, MAC learning for L2 forwarding, or port status monitoring, can be solely performed by the embedded/local controller. Other control plane functions that require an overall view of the network, e.g., routing applications, can be performed by the remote controller, possibly in cooperation with the embedded/local controller where the local controller acts as proxy, i.e., it relays control plane messages between the P4 target and the global controller. Hierarchical control planes improve load distribution as many tasks can be performed locally, which reduces load on the remote controllers. In particular, time-critical operations may benefit from local controllers as additional delays caused by the communication between a P4 target and a global controller are avoided.

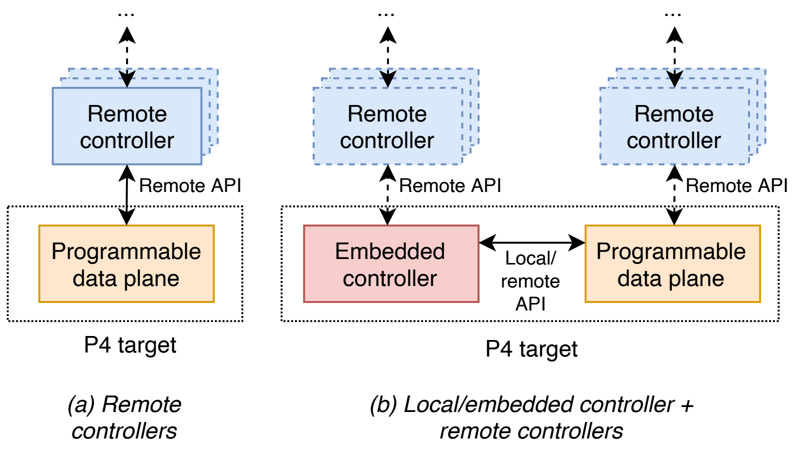


Figure 23: Remote controller use case pattern.

7. Advances in P4 Data Plane Programming

We give an overview on research to improve P4 data plane programming. Figure 24 depicts the structure of this section. We describe related work on optimization of development and deployment, testing and debugging, research on P4 targets, and research on control plane operation.

7.1. Optimization of Development and Deployment

We describe research work on optimizing the development & deployment process of P4.

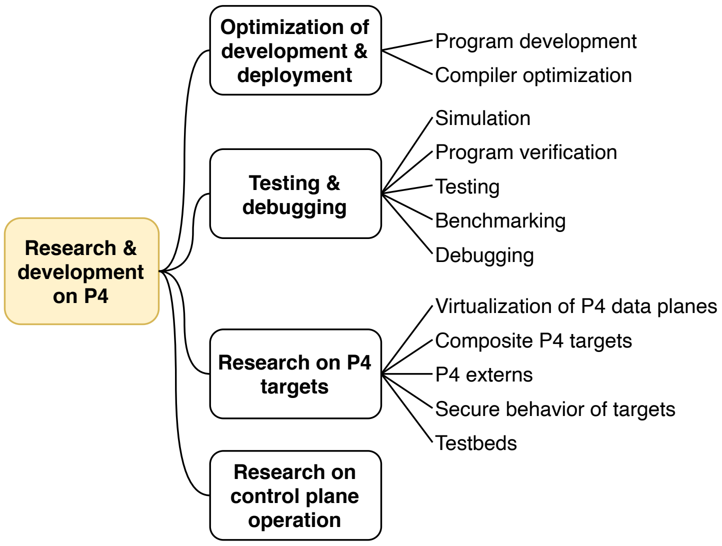


Figure 24: Organization of Section 7.

7.1.1. Program Development

Graph-to-P4 [108] generates P4 program code for given parse graphs. This introduces a higher abstraction layer that is particularly helpful for beginners. Zhou et al. [109] introduce a module system for P4 to improve source code organization. DaPIPE [110] enables incremental deployment of P4 program code on P4 targets. SafeP4 [111] adds type safety to P4. P4I/O [112] presents a framework for intent-based networking with P4. Network operator describe their network functions with an Intent Definition Language (IDL) and P4I/O generates a complete P4 program accordingly. To that end, P4I/O provides a P4 action repository with various network functions. During reconfiguration, table and register state are preserved by applying backup mechanisms. P4I/O is implemented for a custom bmv2. Mantis [113] is a framework to implement fast reactions to changing network conditions in the data plane without controller interaction. To that end, annotations in the P4 code specify dynamic components and a quick control loop of those components ensure timely adjustments if necessary. Lyra [114] is a pipeline abstraction that allows developers to use simple statements to describe their desired data plane without low-level target-specific knowledge. Lyra then compiles that description to target-specific code for execution. GP4P4 [115] is a programming framework for self-driven networks. It generates P4 code from behavioral rules defined by the developer. To that end, GP4P4 evaluates the quality of the automatically generated programs and improves them based on genetic algorithms. FlowBlaze.p4 [116, 117, 118] implements an executor for FlowBlaze, an abstraction based on an extended finite state machine for building stateful packet processing functions, in P4. This library maps FlowBlaze elements to P4 components for execution on the bmv2. It also provides a GUI for defining the extended finite state machine. Flightplan [119] is a programming tool chain that disaggregates a P4 program into multiple P4 programs so that they can be executed on different targets. The authors state that this improves performance, resource utilization, and cost.

7.1.2. Compiler Optimization

pcube [120] is a preprocessor for P4 that translates primitive annotations in P4 programs into P4 code for common operations such as loops. CacheP4 [121] introduces a behavior-level cache in front of the P4 pipeline. It identifies flows and performs a compound of actions to avoid unnecessary table matches. The cache is filled during runtime by a controller that receives notifications from the switch. P5 [122] optimizes the P4 pipeline by removing inter-feature dependencies. dRMT [24] is a new architecture for programmable switches that introduces deterministic throughput and latency guarantees. Therefore, it generates schedules for CPU and memory resources from a P4 program. P2GO [123] leverages monitored traffic information to optimize resource allocation during compilation. It adjusts table and register size to reduce the pipeline length, and offloads rarely used parts of the program to the control plane. Yang et al. [124] propose a compiler module that optimizes lookup speed by reorganizing flow tables and prioritization of popular forwarding rules. Vass et al. [125] analyze and discuss algorithmic aspects of P4 compilation.

7.2. Testing and Debugging

We describe research work on simulation, program verification, testing, benchmarking, and debugging.

7.2.1. Simulation

PFPSim [126] is a simulator for validation of packet processing in P4. NS4 [127, 128] is a network simulator for P4 programs that is based on the network simulator NS3.

7.2.2. Program Verification

McKeown et al. [129] introduce a tool to translate P4 to the Datalog declarative programming language. Then, the Datalog representation of the P4 program can be analyzed for well-formedness. Kheradmand et al. [130] introduce a tool for static analysis of P4 programs that is based on formal semantics. P4v [131] adapts common verification methods for P4 that are based on annotations in the P4 program code. Freire et al. [132, 133] introduce assertion-based verification with symbolic execution. Stoenescu et al. [134] propose program verification based on symbolic execution in combination with a novel description language designed for the properties of P4. P4AIG [135] proposes to use hardware verification techniques where developers have to annotate their code with First Order Logic (FOL) specifications. P4AIG then encodes the P4 program as an Advanced-Inverter-Graph (AIG) which can be verified by hardware verification techniques such as circuit SAT solvers and bounded model checkers. bf4

[136] leverages static code verification and runtime checks of rules that are installed by the controller to confirm that the P4 program is running as intended. netdiff [137] uses symbolic execution to check if two data planes are equivalent. This can be useful to verify if a data plane behaves correctly by comparing it with a similar one, or to verify that optimizations of a data plane do not change its behavior. Yousefi et al. [138] present an abstraction for liveness verification of stateful network functions (NFs). The abstraction is based on boolean formulae. Further, they provide a compiler that translates these formulae into P4 programs.

7.2.3. Testing

P4pktgen [139] generates test cases for P4 programs by creating test packets and table entries. P4Tester [140] implements a detection scheme for runtime faults in P4 programs based on probe packets. P4app [141] is a partially automated open source tool for building, running, debugging, and testing P4 programs with the help of Docker images. P4RL [142] is a reinforcement learning based system for testing P4 programs and P4 targets at runtime. The correct behavior is described in a simple query language so that a reinforcement agent based on Double DQN can learn how to manipulate and generate packets that contradict the expected behavior. P4TrafficTool [143] analyzes P4 programs to produce plugin code for common traffic analyzers and generators such as Wireshark.

7.2.4. Benchmarking

Whippersnapper [144] is a benchmark suite for P4 that differentiates between platform-independent and platform-specific tests. BB-Gen [145] is a system to evaluate P4 programs with existing benchmark tools by translating P4 code into other formats. P8 [146] estimates the average packet latency at compilation time by analyzing the data path program.

7.2.5. Debugging

Kodeswaran et al. [147] propose to use Ball-Larus encoding to track the packet execution path through a P4 program for more precise debugging capabilities. p4-data-flow [148] detects bugs by creating a control flow graph of a P4 program and then identifies incorrect behavior. P4box [149] extends the P4$_{16}$ reference compiler by so-called *monitors* that insert code before and after programmable blocks, e.g., control blocks, for runtime verification. P4DB [150] [151] introduces a runtime debugging system for P4 that leverages additional debugging snippets in the P4 program to generate reports during runtime. Neves et al. [152] propose a sandbox for P4 data plane programs for diagnosis and tracing. P4Consist [153] verifies the consistency between control and data plane. Therefore, it generates active probe-based traffic for which the control and data plane generate independent reports that can be compared later. KeySight [154] is a troubleshooting platform that analyzes network telemetry data for detecting runtime faults. Gauntlet [155] finds both crash bugs, i.e., abnormal termination of compilation operation, and semantic bugs, i.e., miscompilation, in compilers for programmable packet processors.

7.3. Research on P4 Targets

We describe research work on virtualization of P4 data planes, composite targets, P4 externs, secure behavior of targets, and testbeds.

7.3.1. Virtualization of P4 Data Planes

P4 targets are designed to execute one P4 program at any given time. Virtualization aims at sharing the resources of P4 targets for multiple P4 programs. Krude et al. [156] provide theoretical discussions on how ASIC- and FPGA-based P4 targets can be shared between different tenants and how P4 programs can be made hot-pluggable.

HyPer4 [157] introduces virtualization for P4 data planes. It supports scenarios such as network slicing, network snapshotting, and virtual networking. To that end, a compiler translates P4 programs into table entries that configure the HyPer4 *persona*, a P4 program that contains implementations of basic primitives. However, HyPer4 does not support stateful memory (registers, counters, meters), LPM, range match types, and arbitrary checksums. The authors describe an implementation for bmv2 and perform experiments that reveal 80 to 90% lower performance in comparison to native execution.

HyperV [158, 159, 160] is a hypervisor for P4 data planes with modular programmability. It allows isolation and dynamic management of network functions. The authors implemented a prototype for the bmv2 P4 target. In comparison to Hyper4, HyperV achieves a 2.5x performance advantage in terms of bandwidth and latency while reducing required resources by a factor of 4. HyperVDP [161] extends HyperV by an implementation of a dynamic controller that supports instantiating network functions in virtual data planes.

P4VBox [162], also published as VirtP4 [163], is a virtualization framework for the NetFPGA SUME P4 target. It allows executing virtual switch instances in parallel and also to hot-swap them. In contrast to HyPer4, HyperV and HyperVDP, P4VBox achieves virtualization by partially re-configuring the hardware.

P4Visor [164] merges multiple P4 programs. This is done by program overlap analysis and compiler optimization. Programming In-Network Modular Extensions (PRIME) [165] also allows combining several P4 programs to a single program and to steer packets through the specific control flows.

P4click [166] does not only merge multiple P4 programs, but also combines the corresponding control plane blocks. The purpose of P4click is to increase the use of data plane programmability. P4click is currently in an early stage of development.

The Multi Tenant Portable Switch Architecture (MTPSA) [167] is a P4 architecture that offers performance isolation, resource isolation, and security isolation in a switch for multiple tenants. MTPSA is based on the PSA. It combines a *Superuser* pipeline that acts as a hypervisor with multiple user pipelines. User pipelines may only perform specific actions depending on their privileges. MTPSA is implemented for bmv2 and NetFPGA-SUME [168].

Han et al. [169] provide an overview of virtualization in programmable data planes with a focus on P4. They classify virtualization schemes into hypervisor and compiler-based approaches, followed by a discussion of pros and cons of the different schemes. The aforementioned works on virtualization of P4 data planes are described and compared in detail.

7.3.2. Composite P4 Target

Da Silva et al. [170] introduce the idea of composite P4 targets. This tries to solve the problem of target-dependent support of features. The composed data plane appears as one P4 target; it is emulated by a P4 software target but relies on an FPGA and ASIC for packet processing.

eXtra Large Table (XLT) [171] introduces gigabyte-scale MATs by leveraging FPGA and DRAM capabilities. It comprises a P4-capable ASIC and multiple FPGAs with DDR4 DRAM. The P4-capable ASIC pre-constructs the match key field and sends it with the full packet to the FPGA. The FPGA sends back the original packet with the search results of the MAT lookup. The authors implement a DPDK based prototype for the T_4P_4S P4 software target.

HyMoS [172] is a hybrid software and hardware switch to support NFV applications. The authors create a switch by using P4-enabled Smart NICs as line cards and the PCIe interface of a computer as the switch fabric. P4 is used for packet switching between the NICs. Additional processing may be done using DPDK or applications running on a GPU.

7.3.3. P4 Externs

Laki et al. [173, 174] investigate asynchronous execution of externs. In contrast to common synchronous execution, other packets may be processed by the pipeline while the extern function is running. The authors implement and evaluate a prototype for T4P4S. Scholz et al. [175] propose that P4 targets should be extended by cryptographic hash functions that are required to build secure applications and protocols. The authors propose an extension of the PSA and discuss the PoC implementation for a CPU-, network processing unit (NPU)-, and FPGA-based P4 target. Da Silva et al. [176] investigate the implementation of complex operations as extensions to P4. The authors perform a case study on integrating the Robust Header Compression (ROHC) scheme and conclude that an implementation as extern function is superior to an implementation as a new native primitive.

7.3.4. Secure Behaviour of Targets

Gray et al. [177] demonstrate that hardware details of P4 targets influence their packet processing behavior. The authors demonstrate this by sending a special traffic pattern to a P4 firewall. It fills the cache of this target and results in a blocking behavior although the overall data rate is far below the capacity of the used P4 target. Dumitru et al. [178] investigate the exploitation of programming bugs in bmv2, P4-NetFPGA, and Tofino. The authors demonstrate attack scenarios by header field access on invalid headers, the creation of infinite loops and unintentionally processing of dropped packets in the P4 targets.

7.3.5. Testbeds

Large testbeds facilitate research and development on P4 programs. The i-4PEN (International P4 Experimental Networks) [179] is an international P4 testbed operated by a collaboration of network research institutions from the USA, Canada, and Taiwan. Chung et al.[180] describe how multi-tenancy is achieved in this testbed. The 2STiC testbed [181], a national testbed in the Netherlands comprising six sites with at least one Tofino-based P4 target, is connected to i-4PEN.

7.4. Research on Control Plane Operation

When new forwarding entries are computed by the controller, the data plane has to be updated. However, updating the targets has to be performed in a manner that prevents negative side effects. For example, microloops may occur if packets are forwarded according to new rules at some targets while at other devices old rules are used because updates have to arrive yet.

Sukapuram et al. [182, 183] introduce a timestamp in the packet header that contains the sending time of a packet. When switches receive a packet during an update period, they compare the timestamp of both the packet and the update to determine whether a packet has been sent before the update, and thus, old rules should be used for forwarding.

Liu et al. [184] introduce a mechanism where once a packet is matched against a specific forwarding rule, it cannot be matched downstream on a rule that is older. To that end, the packet header contains a timestamp field that records when the last applied forwarding rule has been updated. If the packet is matched against an older rule, the packet is dropped, otherwise the timestamp is updated and the packet is forwarded.

Ez-Segway [185] facilitates updating by including data plane devices in the update process. When a data plane device receives an update, it determines which of its neighbors is affected by the update as well, and forwards the update to that neighbor. This prevents loops and black holes.

TableVisor [186] is a transparent proxy-layer between the control plane and data plane. It provides an abstraction from heterogeneous data plane devices. This facilitates the configuration of data plane switches with different properties, e.g., forwarding table size.

Molero et al. [187] propose to offload tasks from the control plane to the data plane. They show that programmable data planes are able to run typical control plane operations like failure detection and notification, and connectivity retrieval. They discuss trade-offs, limitations and future research opportunities.

8. Applied Research Domains: Classification & Overview

In the following sections, we give an overview of applied research conducted with P4. In this section, we classify P4's core features that make it attractive for the implementation of data plane algorithms. We define research domains, visualize them in a compact way, and explain our method to review corresponding research papers in the subsequent sections. Finally, we delimit the scope of the surveyed literature.

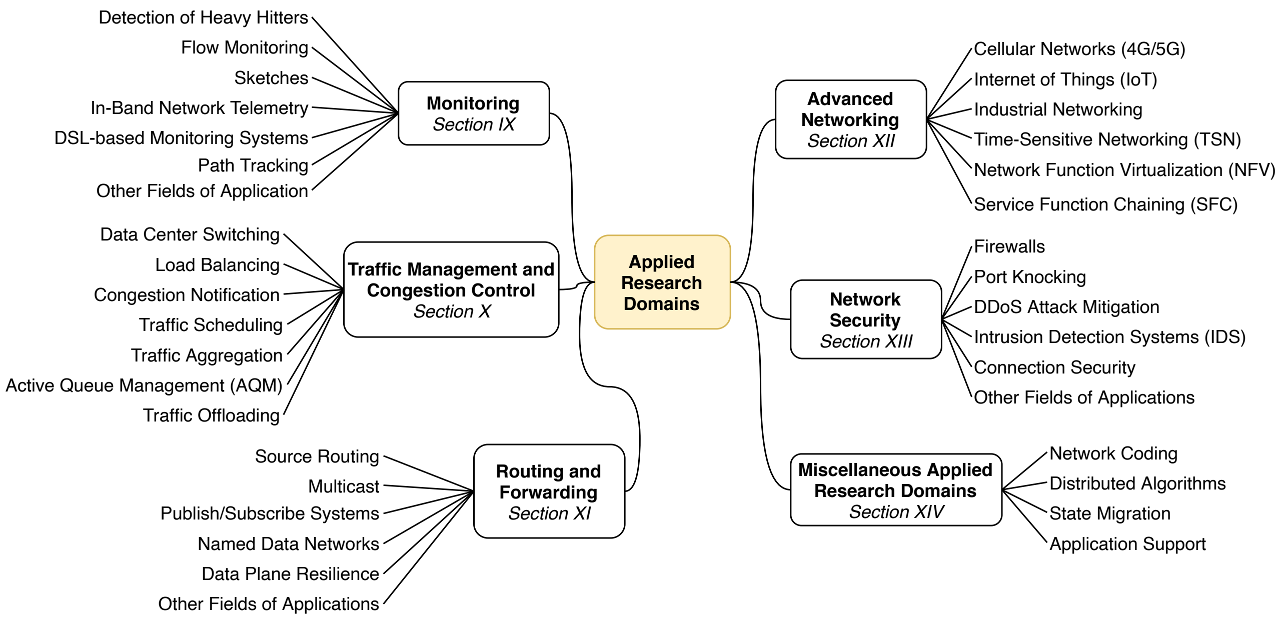

Applied Research Domains
Monitoring Section IX
Detection of Heavy Hitters
Flow Monitoring
Sketches
In-Band Network Telemetry
DSL-based Monitoring Systems
Path Tracking
Other Fields of Application
Traffic Management and Congestion Control Section X
Data Center Switching
Load Balancing
Congestion Notification
Traffic Scheduling
Traffic Aggregation
Active Queue Management (AQM)
Traffic Offloading
Routing and Forwarding Section XI
Source Routing
Multicast
Publish/Subscribe Systems
Named Data Networks
Data Plane Resilience
Other Fields of Applications
Advanced Networking Section XII
Cellular Networks (4G/5G)
Internet of Things (IoT)
Industrial Networking
Time-Sensitive Networking (TSN)
Network Function Virtualization (NFV)
Service Function Chaining (SFC)
Network Security Section XIII
Firewalls
Port Knocking
DDoS Attack Mitigation
Intrusion Detection Systems (IDS)
Connection Security
Other Fields of Applications
Miscellaneous Applied Research Domains Section XIV
Network Coding
Distributed Algorithms
State Migration
Application Support

Figure 25: Categorization of the surveyed works into applied research domains and subdomains – they correspond to sections and subsections in the remainder of this paper.

8.1. Classification of P4's Core Features

We identify P4's core features for the implementation of prototypes. We classify them in the following to effectively reason about P4's usefulness for the surveyed research works.

8.1.1. Definition and Usage of Custom Packet Headers

P4 requires the definition of packet headers (Section 3.5). These may be headers of standard protocols, e.g., TCP, use-case-specific protocols, e.g., GTP in 5G, or new protocols. As P4 supports the definition of custom headers, it is suitable for the implementation of data plane algorithms using new protocols or extensions of existing protocols, e.g., for in-band signalling.

8.1.2. Flexible Packet Header Processing

Control blocks with MATs (Section 3.6) comprise the packet processing logic. Packet processing includes default actions, e.g., forwarding and header field modifications, or custom, user-defined actions. Both may be parameterized via MATs or metadata. Entries in the MATs are maintained by a data plane API (Section 6). The flexible use of actions, the definition of new actions, and their parameterization offer high flexibility for header processing, which is often needed for research prototypes.

8.1.3. Target-Specific Packet Header Processing Functions

While the above-mentioned features are part of the P4 core language and supported by any P4-capable platform, devices may offer additional architecture- or target-specific functionality which is made available as *P4 extern* (Section 4). Typical externs include components for stateful processing, e.g., registers or counters, operations to resubmit/recirculate the packet in the data plane, multicast operations, or more complex operations, e.g., hashing and encryption/decryption. P4 software targets allow users to integrate custom externs and use them within P4 programs. While this is also possible to some extent on some P4 hardware targets, e.g., the NetFPGA SUME board, high-throughput P4 targets based on the Tofino ASIC have only a fixed set of externs (Section 5.3). Depending on the use case, the availability of externs may be essential for the implementation of prototypes. Thus, externs facilitate the implementation of more complex algorithms but make implementations platform-dependent.

8.1.4. Packet Processing on the Control Plane

Similar to control plane SDN (e.g., OF), more complex, and optionally centralized packet processing can be outsourced to an SDN control plane; packet exchange and data plane control is performed via a data plane API (Section 6). While OF only allows the exchange of complete packets, P4 enables the end-users to define the packet formats.

8.1.5. Flexible Development and Deployment

Users are able to easily change the P4 programs on P4 targets that are installed in a network. This facilitates agile development with frequent deployments and incremental functionality extensions by deploying new versions of a P4 programs.

8.2. Categorization of Research Domains

To organize the survey in the following sections, we define research domains and structure them in a two-level hierarchy as depicted in Figure 25. This categorization helps the reader to get a quick overview in certain applied areas and improves the readability of this survey. The choice of the research domains is dominated by the fields of applications, but the summaries of the sections will show that the prototypes in these areas benefit from different core features of P4.

For each research domain, we provide a table that lists the publications with publication year, P4 target platforms, and source code availability. This supports efficient browsing of the content and backs our conclusions in the section-specific summaries.

8.3. Scope of the Surveyed Literature

We consider the literature until the end of 2020 and selected papers from 2021, including journal papers, conference papers, workshop papers, and preprints. Out of the 377 scientific publications we surveyed in this work (see Section 1), 245 fall in the area of applied research. 68 of those research papers were published in 2018 or before, 80 were published in 2019, 93 were published in 2020, and 4 were published in 2021. 60 out of all 245 research publications released the source code of their prototype implementations.

Table 4 depicts a statistic on major publication venues for the papers of applied research domains. It helps the reader to identify potential venues for prospective own publications based on P4 technology.

9. Applied Research Domains: Monitoring

We describe applied research on detection of heavy hitters, flow monitoring, sketches, in-band network telemetry, and other areas of application. Table 5 shows an overview of all the work described. At the end of the section, we summarize the work and analyze it with regard to P4's core features described in Section 8.1.

9.1. Detection of Heavy Hitters

Heavy hitters [268] (or "elephant flows") are large traffic flows that are the major source of network congestion. Detection mechanisms aim at identifying heavy hitters to perform extra processing, e.g., queuing, flow rate control, and traffic engineering.

HashPipe [188] integrates a heavy hitter detection algorithm entirely on the P4 data plane. A pipeline of hash tables acts as a counter for detected flows. To fulfill memory constraints, the number of flows that can be stored is limited. When a new flow is detected, it replaces the flow with the lowest count. Thus, light flows are replaced, and heavy flows can be detected by a high count. Lin et al. [190] describe an enhanced version of the algorithm.

Popescu et al. [191] introduce a heavy hitter detection mechanism. The controller installs TCAM entries for specific source IP prefixes on the switch. If one of these entries matches more often than a threshold during a given time frame, the entry is split into two entries with a larger prefix size. This procedure is repeated until the configured granularity is reached.

Harrison et al. [192] presents a controller-based and distributed detection scheme for heavy hitters. The authors make use of counters for the match key values, e.g., source and destination IP pair or 5-tuple, that are maintained by P4 switches. If a counter exceeds a certain threshold, the P4 switch sends a notification to the controller. The controller generates more accurate status reports by combining the notifications received from the switches.

Kucera et al. [193] describe a system for detecting traffic aggregates. The authors propose a novel algorithm that supports hierarchical heavy hitter detection, change detection, and super-spreader detection. The complete mechanism is implemented on the P4 data plane and uses push notifications to a controller.

Table 4: Statistics of scientific publications regarding applied research conducted with P4.

Venue	**#Publications**
Journals	**41**
IEEE ACCESS	9
IEEE/ACM ToN	7
IEEE TNSM	6
JNCA	4
Miscellaneous	15
Conferences	**168**
ACM SOSR	14
IEEE NFV-SDN	12
IEEE ICNP	12
IEEE ICC	10
ACM SIGCOMM	10
IEEE/IFIP NOMS	8
ACM CoNEXT	7
IEEE NetSoft	7
USENIX NSDI	6
IEEE INFOCOM	6
ACM/IEEE ANCS	5
IFIP Networking	5
IEEE GLOBECOM	4
CNSM	4
IEEE CloudNet	3
APNOMS	3
IFIP/IEEE IM	3
Miscellaneous	49
Workshops	**36**
EuroP4	11
Morning Workshop on In-Network Computing	5
SPIN	3
ACM HotNets	3
INFOCOM Workshops	3
Miscellaneous	11

Table 5: Overview of applied research on monitoring (Section 9).

Research work	**Year**	**Targets**	**Code**
Detection of Heavy Hitters (Section 9.1)			
HashPipe [188]	2017	bmv2	[189]
Lin et al. [190]	2019	Tofino	
Popescu et al. [191]	2017	-	
Harrison et al. [192]	2018	Tofino	
Kucera et al. [193]	2020	bmv2	
IDEAFIX [194]	2018	-	
Turkovic et al. [195]	2019	Netronome	
Ding et al. [196]	2020	bmv2	[197]
Flow Monitoring (Section 9.2)			
TurboFlow [198]	2018	Tofino, Netronome	[199]
∗Flow [200]	2018	Tofino	[201]
Hill et al. [202]	2018	bmv2	
FlowStalker [203]	2019	bmv2	
ShadowFS [204]	2020	bmv2	
FlowLens [205]	2021	bmv2, Tofino	[206]
SpiderMon [207]	2020	bmv2	
ConQuest [208]	2019	Tofino	
Zhao et al. [209]	2019	bmv2, Tofino	
Sketches (Section 9.3)			
SketchLearn [210]	2018	Tofino	[211]
MV-Sketch [212]	2020	bmv2, Tofino	[213]
Hang et al. [214]	2019	Tofino	
UnivMon [215]	2016	p4c-behavioural	
Yang et al. [216, 217]	2018/19	Tofino	[218]
Pereira et al. [219]	2017	bmv2	
Martins et al. [220]	2018	bmv2	
Lai et al. [221]	2019	Tofino	
Liu et al. [222]	2020	Tofino	
SpreadSketch [223]	2020	Tofino	[224]

Research work	Year	Targets	Code
In-Band Network Telemetry (Section 9.4)			
Vestin et al. [225]	2019	Netronome	
Wang et al. [226]	2019	Tofino	
IntOpt [227]	2019	P4FPGA	
Jia et al. [228]	2020	bmv2	[229]
Niu et al. [230]	2019	Tofino, Netronome	
CAPEST [231]	2020	bmv2	[232]
Choi et al. [233]	2019	bmv2	
Sgambelluri et al. [234]	2020	bmv2	
Feng et al. [235]	2020	Netronome	
IntSight [236]	2020	bmv2, NetFPGA-SUME	[237]
Suh et al. [238]	2020	-	
DSL-Based Monitoring Systems (Section 9.5)			
Marple [239, 240]	2017	bmv2	[241]
MAFIA [242]	2019	bmv2	[243]
Sonata [244]	2018	bmv2, Tofino	[245]
Teixeira et al. [246]	2020	bmv2, Tofino	
Path Tracking (Section 9.6)			
UniRope [247]	2018	bmv2, PISCES	
Knossen et al. [248]	2019	Netronome	
Basuki et al. [249]	2020	bmv2	
Other Areas of Application (Section 9.7)			
BurstRadar [250]	2018	Tofino	[251]
Dapper [252]	2017	-	
He et al. [253]	2018	Tofino	
Riesenberg et al. [254]	2019	bmv2	[255]
Wang et al. [256]	2020	Tofino	
P4STA [257]	2020	bmv2, Netronome	[258]
Hark et al. [259]	2019	-	
P4Entropy [260]	2020	bmv2	[261]
Taffet et al. [262]	2019	bmv2	
NetView [263]	2020	bmv2, Tofino	
FastFE [264]	2020	Tofino	
Unroller [265]	2020	bmv2, Netcope P4-to-VHDL	
Hang et al. [266]	2019	Tofino	
FlowSpy [267]	2019	bmv2	

IDEAFIX [194] is a system that detects elephant flows at edge switches of Internet exchange point networks. The proposed system analyzes flow features, stores them with hash keys as indices in P4 registers, and compares them to thresholds for classification.

Turkovic et al. [195] propose a streaming approach for detecting heavy hitters via sliding windows that are implemented in P4. According to the authors, interval methods that are typically used to detect heavy hitters are not suitable for programmable data planes because of high hardware resources, bad accuracy, or a need for too much intervention by the control plane.

Ding et al. [196] propose an architecture for network-wide heavy hitter detection. The authors' main focuses are hybrid SDN/non-SDN networks where programmable devices are deployed only partially. To that end, they also present an algorithm for an incremental deployment of programmable devices with the goal of maximizing the number of network flows that can be monitored.

9.2. Flow Monitoring

In flow monitoring, traffic is analyzed on a per-flow level. Network devices are configured to export per-flow information, e.g., packet counters, source and target IP addresses, ports, or protocol types, as flow records to a flow collector. These flow records are often duplicates of network packets without payload data. The flow collector then performs centralized analysis on this data. The three most widely deployed protocols are Netflow [269], sFlow [270], and IPFIX [271].

TurboFlow [198] is a flow record generator designed for P4 switches that does not have to make use of sampling or mirroring. The data plane generates micro-flow records with information about the most recent packets of a flow. On the CPU module of the switch, those micro-flow records are aggregated and processed into full flow records.

"$*$Flow" [200] partitions measurement queries between the data plane and a software component. A switching ASIC computes grouped packet vectors that contain a flow identifier and a variable set of packet features, e.g. packet size and timestamps, while the software component performs aggregation. "$*$Flow" supports dynamic and concurrent measurement applications, i.e., measurement applications that operate on the same flows without impacting each other.

Hill et al. [202] implement Bloom filters on P4 switches to prevent sending duplicate flow samples. Bloom filters are a probabilistic data structure that can be used to check whether an entry is present in a set or not. It is possible to add elements to that set, but it is not possible to remove entries from it. For flow tracking, Bloom filters test if a flow has been seen before without control plane interaction. Thereby, only flow data is forwarded to the collector from flows that were not seen before.

FlowStalker [203] is a flow monitoring system running on the P4 data plane. The monitoring operations on a packet are divided in two phases, a proactive phase that identifies a flow and keeps a per-flow packet counter and a reactive phase that runs for large flows only and gathers metrics of the flow, e.g., byte counts and packet sizes. The controller gathers information from a cluster of switches by injecting a crawler packet that travels through the cluster at one switch. ShadowFS [204] extends FlowStalker with a mechanism to increase the throughput of the monitored flows. It achieves this by dividing forwarding tables into two tables, a faster and a slower one. The most utilized flows are moved to the faster table if necessary.

FlowLens [205] is a system for traffic classification to support security network applications based on machine learning algorithms. The authors propose a novel memory-efficient representation for features of flows called *flow marker*. A profiler running in the control plane automatically generates an application-specific flow marker that optimizes the trade-off between resource consumption and classification accuracy, according to a given criterion selected by the operator.

SpiderMon [207] monitors network performance and debugs performance failures inside the network with little overhead. To that end, SpiderMon monitors every flow in the data plane and recognizes if the accumulated latency exceeds a certain threshold. Furthermore, SpiderMon is able to trace back the path of interfering flows, allowing to analyze the cause of the performance degradation.

ConQuest [208] is a data plane mechanism to identify flows that occupy large portions of buffers. Switches maintain snapshots of queues in registers to determine the contribution to queue occupancy of the flow of a received packet.

Zhao et al. [209] implement flow monitoring using hash tables. Using a novel strategy for collision resolution and record promotion, accurate records for elephant flows and summarized records for other flows are stored.

9.3. Sketches

Flow monitoring as described in Section 9.2 requires high sampling rates to produce sufficiently detailed data. As an alternative, streaming algorithms process sequential data streams and are subject to different constraints like limited memory or processing time per item. They approximate the current network status based on concluded summaries of the data stream. The streaming algorithms output so-called sketches that contain summarized information about selected properties of the last n packets of a flow.

SketchLearn [210] is a sketch-based approach to track the frequency of flow records. It features multilevel sketches that aim for small memory usage, fast per-packet processing, and real-time response. Rather than finding the perfect resource configuration for measurement traffic and regular traffic, SketchLearn characterizes the statistical error of resource conflicts based on Gaussian distributions. The learned properties are then used to increase the accuracy of the approximated measurements.

Tang et al. [212] present MV-Sketch, a fast and compact invertible sketch. MV-Sketch leverages the idea of majority voting to decide whether a flow is a heavy hitter or heavy changer. Evaluations show that MV-Sketch achieves a 3.38 times higher throughput than existing invertible sketches.

Hang et al. [214] try to solve the problem of inconsistency when a controller needs to collect the data from sketches on one or more switches. As accessing and clearing the sketches on the switches is always subject to latency, not all sketches are reset at the same time, and there might be some delay between accessing and clearing the sketches. The authors propose to use two asymmetric sketches on the switches that are used in an interleaved way. Furthermore, the authors propose to use a distributed control plane to keep latency low.

UnivMon [215] is a flow monitoring system based on sketches. After sampling the traffic, the data plane produces sketches and determines the top-k heaviest flows by comparing the number of sketches for each flow. Those flows are passed to the control plane which processes the data for the specific application.

Yang et al. [216, 217] propose to adapt sketches according to certain traffic characteristics to increase data accuracy, e.g., during congestion or distributed denial of service (DDoS) attacks. The mechanism is based on compressing and merging sketches when resources in the network are limited due to high traffic volume. During periods with high packet rates, only the information of elephant flows is recorded to trade accuracy for higher processing speed.

Pereira et al. [219] propose a secured version of the Count-Min sketch. They replace the function with a cryptographic hash function and provide a way for secret key renewal.

Martins et al. [220] introduce sketches for multi-tenant environments. The authors implement bitmap and counter-array sketches using a new probabilistic data structure called BitMatrix that consists of multiple bitmaps that are stored in a single P4 register.

Lai et al. [221] use a sketch-based approach to estimate the entropy of network traffic. The authors use CRC32 hashes of header fields as match keys for match-action tables and subsequently update k-dimensional data sketches in registers. The content of the registers is then processed by the control plane CPU which calculates the entropy value.

Liu et al. [222] use sketches for performance monitoring. They introduce lean algorithms to measure metrics like loss or out-of-order packets.

SpreadSketch [223] is a sketch data structure to detect superspreaders. The sketch data structure is invertible, i.e., it is possible to extract the identification of superspreaders from the sketch at the end of an epoch.

9.4. In-Band Network Telemetry

Barefoot Networks, Arista, Dell, Intel and VMware specified in-band network telemetry (INT) specifically for P4 [272]. It uses a pure data plane implementation to collect telemetry data from the network without any intervention by the control plane. It was specified by INT is the main focus of the *Applications WG* [273] of the P4 Language Consortium. Instructions for INT-enabled devices that serve as traffic sources are embedded as header fields either into normal packets or into dedicated probe packets. Traffic sinks retrieve the results of instructions to traffic sources. In this way, traffic sinks have access to information about the data plane state of the INT-enabled devices that forwarded the packets containing the instructions for traffic sources. The authors of the INT specification name network troubleshooting, advanced congestion control, advanced routing, and network data plane verification as examples for high-level use cases.

In two demos, INT was used for diagnosing the cause of latency spikes during HTTP transfers [274] and for enforcing QoS policies on a per-packet basis across a metro network [275].

Vestin et al. [225] enhance INT traffic sinks by event detection. Instead of exporting telemetry items of all packets to a stream processor, exporting has to be triggered by an event. Furthermore, they implement an INT report collector for Linux that can stream telemetry data to a Kafka cluster.

Wang et al. [226] design an INT system that can track which rules in MATs matched on a packet. The resulting data is stored in a database to facilitate visualization in a web UI.

IntOpt [227] uses INT to monitor service function chains. The system computes minimal monitoring flows that cover all desired telemetry demands, i.e., the number of INT-sources, sinks, and forwarding nodes that are covered by this flow is minimal. IntOpt uses active probing, i.e., monitoring probes for the monitoring flows are periodically inserted into the network.

Jia et al. [228] use INT to detect gray failures in data center networks using probe packets. Gray failures are failures that happen silently and without notification.

Niu et al. [230] design a multilevel INT system for IP-over-optical networks. Their goal is to monitor both the IP network and the optical network at the same time. To that end, they implement optical performance monitors for bandwidth-variable wavelength selective switches. Their measurements can be queried by a P4 switch that is connected directly to it.

CAPEST [231] leverages P4-enabled switches to estimate the network capacity and available bandwidth of network links. The approach is passive, i.e., it does not disturb the network. A controller sends INT probe packets to trigger statistical analysis and export results.

Choi et al. [233] leverage INT for run-time performance monitoring, verification, and healing of end-to-end services. P4-capable switches monitor the network based on INT information and the distributed control plane verifies that SLAs and other metrics are fulfilled. They leverage metric dynamic logic (MDL) to specify formal assertions for SLAs.

Sgambelluri et at. [234] propose a multi-layer monitoring system that uses an OpenConfig NETCONF agent for the optical layer an P4-based INT for the packet layer. In their prototype, they use INT to measure the delay of packets by computing the processing time at each switch.

Feng et al. [235] implement an INT sink for Netronome Smart NICs. After parsing the INT headers using P4, they use algorithms written in C to perform INT tasks like aggregation and notification. Compared to a pure P4 implementation, this increases the performance.

IntSight [236] is a system for detecting and analyzing violations of service-level objects (SLOs). SLOs are performance guarantees towards a network, e.g., concerning bandwidth and latency. IntSight uses INT to monitor the performance of the network during a specific period of time. Egress devices gather this information and produce a report at the end of the period if an SLO has been violated.

Suh et al. [238] explore how a sampling mechanism can be added to INT. Their solution supports rate-based and event-based sampling. Based on these sampling strategies, INT headers are only added to a fraction of the packets to reduce overhead.

9.5. DSL-Based Monitoring Systems

Monitoring tasks can often be broken down in a set of several basic operations, e.g., map, filter, or groupby. A domain-specific language (DSL) allows to combine these basic operations in more complex tasks.

Marple [239, 240] is a performance query language that supports existing constructs like map, filter, groupby, and zip. A query compiler translates the queries either to P4 or to a simulator for programmable switch hardware. Stateless constructs of the query language, e.g., filters, are executed on the data plane. Stateful constructs, e.g., groupby, use a programmable key-value store that is split between a fast on-chip SRAM cache and a large off-chip DRAM backing store. The results are streamed from the switch to a collection server.

MAFIA [242] is a DSL to describe network measurement tasks. They identify several fundamental primitive operations, examples are match, tag, timestamp, sketch, or counter. MAFIA is a high-level language to describe more complex measurement tasks composed of those primitives. The authors provide a Python-based compiler that translates MAFIA code into a P4 program in $P4_{14}$ or $P4_{16}$ for a PISA-based P4 target.

Sonata [244] is a query-driven telemetry system. It provides a query interface that provides common operators like map and reduce that can be applied on arbitrary packet fields. Sonata combines the capabilities of both programmable switches and stream processors. The queries are partitioned between the programmable switches and the stream processors to reduce the load on the stream processors. Teixeira et al. [246] extend the Sonata prototype by functionalities to monitor the properties of packet processing inside switches, e.g., delay.

9.6. Path Tracking

In path tracking, or packet trajectory tracing, information about the path a packet has taken in a network is gathered.

UniRope [247] consists of two different algorithms for packet trajectory tracing that can be selected dynamically to be able to choose the trade-off between accuracy and efficiency. These two algorithms are *compact hash matching* and *consecutive bits filling*. With compact hash matching, the forwarding switch calculates a hash value and stores it in the packet. With consecutive bits filling, the packet trajectory is recorded in the packet hop by hop and reconstructed at the controller.

Knossen et al. [248] present two different approaches for path tracking in P4. In *hop recording*, all forwarding P4 nodes record their ID in the header of the target packet. The last node can then reconstruct the path. In *forwarding state* *logging*, the first P4 node records the current version of the global forwarding state of the network and its node identifier in a header of the target packet. If the version of the global forwarding state does not change while the packet flows through the network, the last P4 node in the network can reconstruct the path using the information in the header.

Basuki et al. [249] propose a privacy-aware path-tracking mechanism. Their goal is that the trajectory information in the packets cannot be used to draw conclusions about the network topology or routing information. They achieve this by recording the information in an in-packet bloom filter.

9.7. Other Fields of Application

BurstRadar [250] is a system for microburst detection for data center networks that runs directly on P4 switches. If queue-induced delay is above a certain threshold, BurstRadar reports a microburst and creates a snapshot of the telemetry information of involved packets. This telemetry information is then forwarded to a monitoring server. As it is not possible to gather telemetry information of packets that are already part of the egress queue, the telemetry information of all packets and their corresponding egress port are temporarily stored in a ring buffer that is implemented using P4 registers.

Dapper [252] is a P4 tool to evaluate TCP. It implements TCP in P4 and analyzes header fields, packets sizes, and timestamps of data and ACK packets to detect congestion. Then, flow-dependent information are stored in registers.

He et al. [253] propose an adaptive expiration timeout mechanism for flow entries in P4 switches. The switches implement a mechanism to detect the last packet of a TCP flow. In case of a match, it notifies the controller to delete the corresponding flow entries.

Riesenberg et al. [254] implement alternate marking performance measurement (AM-PM) for P4. AM-PM measures delay and packet loss in-band in a network using only one or two bit overhead per packet. These bits are used for coordination and signalling between measurement points (MPs).

Wang et al. [256] describe how TCP-friendly meters can be designed and implemented for P4-based switches. According to their findings, meters in commercial switches interact with TCP streams in such a way that these streams can only reach about 10% of the target rate. The experimental evaluation of their TCP-friendly meters shows achieved rates of up to 85% of the target rate.

P4STA [257] is an open-source framework that combines software-based traffic load generation with accurate hardware packet timestamps. Thereby, P4STA aggregates multiple traffic flows to generate high traffic load and leverage programmable platforms.

Hark et al. [259] use P4 to filter data plane measurements. To save resources, only relevant measurements are sent to the controller. The authors implement a prototype and demonstrate the system by filtering measurements for a bandwidth forecast application.

P4Entropy [260] presents an algorithm to estimate the entropy of network traffic within the P4 data plane. To that end, they also developed two new algorithms, P4Log and P4Exp, to estimate logarithms and exponential functions within the data plane as well.

Taffet et al. [262] describe a P4-based implementation of an in-band monitoring system that collects information about the path of a packet and whether it encountered congestion. For this purpose, the authors repurpose previously unused fields of the IP header.

NetView [263] is a network telemetry framework that uses proactive probe packets to monitor devices. Telemetry targets, frequency, and characteristics can be configured on demand by administrators. The probe packets traverse arbitrary paths by using source routing.

FastFE [264] is a system for offloading feature extraction, i.e., deriving certain information from network traffic, for machine learning (ML)-based traffic analysis applications. Policies for feature extraction are defined as sequential programs. A policy enforcement engine translates these policies into primitives for either a programmable switch or a program running on a commodity server.

Unroller [265] detects routing loops in the data plane in real-time. It achieves this by encoding a subset of the path that a packet takes into the packet.

Hang et al. [266] use a time-based sliding window approach to measure packet rates. The goal is to record statistics entirely inside the data plane without having to use the CPU of a switch. Their approach is able to measure traffic size without sampling.

FlowSpy [267] is a network monitoring framework that uses load balancing. Different monitoring tasks are distributed among all available switches by an ILP solver. This reduces the workload on single switches in contrast to monitoring frameworks that perform all monitoring tasks on ingress or egress switches only.

9.8. Summary and Analysis

This research domain greatly benefits from all five core features described in Section 8.1. *Definition and usage of custom packet headers* enables new monitoring schemes where relevant information can be added to packets while it travels through a P4-enabled network. One example is In-band Network Telemetry (INT) (Section 9.4) that has been specified specifically for P4. Another example are path tracking mechanisms (Section 9.6) where the path of a packet is recorded in a dedicated header of the packet. In the case of INT, this goes hand in hand with *flexible packet header processing* as INT headers may contain instructions that other INT-enabled switches need to execute. *Target-specific packet header processing functions* in the form of stateful packet processing using, e.g., registers, is used by all areas of monitoring as it is necessary to gather data over a certain time frame instead of just looking at a single packet. Because the register space is severely limited on most hardware targets, an efficient usage of the available resources is of great importance. Sketches (Section 9.3) is one approach to solve this. After monitoring data is gathered on the control plane, the result is often *processed on the control plane.* This can range from simple notifications to splitting operations between data plane and control plane where the resources on the data plane are not sufficient. Some DSL-based monitoring approaches (Section 9.5) make use of *flexible development and deployment.*

With these approaches, a P4 program is generated automatically on the basis of a monitoring workflow defined by an administrator.

10. Applied Research Domains: Traffic Management and Congestion Control

We describe applied research on data center switching, load balancing, congestion notification, traffic scheduling, traffic aggregation, active queue management (AQM), and traffic offloading. Table 6 shows an overview of all the work described. At the end of the section, we summarize the work and analyze it with regard to P4's core features described in Section 8.1.

10.1. Data Center Switching

Trellis [276, 277] is an open-source multipurpose L2/L3 spine-leaf switch fabric for data center networks. It is designed to run on white box switches in conjunction with the ONOS controller where its main functionality is implemented. It supports typical data center functionality such as bridging using VLANs, routing (IPv4/IPv6 unicast/multicast routing, MPLS segment routing), and vRouter functionality (BGBv4/v6, static routes, route black-holing). Trellis is part of the CORD platform that leverages SDN, network function virtualization (NFV), and Cloud technologies for building agile data centers for the network edge.

DC.p4 [279] implements typical features of data center switches in P4. The list of features includes support for VLAN, NVGRE, VXLAN, ECMP, IP forwarding, access control lists (ACLs), packet mirroring, MAC learning, and packet-in/-out messages to the control plane.

Fabric.p4 is [281, 277] the underlying reference data plane pipeline implemented in P4. By introducing support for P4 switches, the authors aim at increasing the platform heterogeneity for the CORD fabric. Fabric.p4 is currently based on the V1Model switch architecture, but support for PSA is planned. It is inspired by the OpenFlow data plane abstraction (OF-DPA) and currently supports L2 bridging, IPv4/IPv6 unicast/multicast routing, and MPLS segment routing. Fabric.p4 comes with capability profiles such as *fabric* (basic profile), *spgw* (S/PGW), and INT. For control plane interaction, ONOS is extended by the P4Runtime.

RARE [283] (Router for Academia, Research & Education) is developed in the GÉANT project GN4-3 and implements a P4 data plane for the FreeRouter open-source control plane. Its feature list includes routing, bridging, ACLs, VLAN, VXLAN, MPLS, GRE, MLDP, and BIER among others.

10.2. Load Balancing

SHELL [285] implements stateless application-aware load balancing in P4. A load balancer forwards new connections to a set of randomly chosen application instances by adding a segment routing (SR) header. Each application instance makes a local decision to either decline or accept the connection attempt. After

Table 6: Overview of applied research on traffic management and congestion control (Section 10).

Research work	Year	Targets	Code
Data Center Switching (Section 10.1)			
Trellis [276, 277]	2019	bmv2	[278]
DC.p4 [279]	2015	bmv2	[280]
Fabric.p4 [281]	2018	bmv2	[282]
RARE [283]	2019	bmv2, Tofino	[284]
Load Balancing (Section 10.2)			
SHELL [285]	2018	NetFPGA-SUME	
SilkRoad [286]	2017	Tofino	
HULA [287]	2016	-	
MP-HULA [288]	2018	-	
Chiang et al. [289]	2019	bmv2	
W-ECMP [290]	2018	bmv2	
DASH [291]	2020	bmv2	
Pizzutti et al. [292, 293]	2018/20	bmv2	
LBAS [294]	2020	Tofino	
DPRO [295]	2020	bmv2	
Kawaguchi et al. [296]	2019	bmv2	
AppSwitch [297]	2017	PISCES	
Beamer [298]	2018	bmv2, NetFPGA-SUME	[299]
Congestion Notification (Section 10.3)			
P4QCN [300]	2019	bmv2	
Jiang et al. [301]	2019	-	
EECN [302]	2020	bmv2	
Chen et al. [303]	2020	bmv2	
Laraba et al. [304]	2020	bmv2	
Traffic Scheduling (Section 10.4)			
Sharma et al. [305]	2018	bmv2	
Cascone et al. [306]	2017	-	
Bhat et al. [307]	2019	bmv2	
Kfoury et al. [308]	2019	bmv2	
Chen et al. [309]	2019	Tofino	
Lee et al. [310]	2019	bmv2	
Traffic Aggregation (Section 10.5)			
Wang et al. [311]	2020	Tofino	
RL-SP-DRR [312]	2019	bmv2	

Research work	**Year**	**Targets**	**Code**
Active Queue Management (AQM) (Section 10.6)			
Turkovic et al. [313]	2018	bmv2, Netronome	
P4-Codel [314]	2018	bmv2	[315]
P4-ABC [316]	2019	bmv2	
P4air [317]	2020	bmv2, Tofino	
Fernandes et al. [318]	2020	bmv2	
Wang et al. [319]	2018	bmv2, Tofino	
SP-PIFO [320]	2020	Tofino	
Kunze et al. [321]	2021	Tofino	[322]
Harkous et al. [323]	2021	bmv2, Netronome	
Traffic Offloading (Section 10.7)			
Andrus et al. [324]	2019	-	
Ibanez et al. [325]	2019	NetFPGA-SUME	
Kfoury et al. [326]	2020	Tofino	
Falcon [327]	2020	Tofino	
Osiński et al. [328]	2020	Tofino	

connection initiation, the client includes a previously negotiated identifier in all subsequent packets. In the prototypical implementation, the authors use TCP time stamps for communicating the identifier, alternatives are identifiers of QUIC or TCP sequence numbers.

SilkRoad [286] implements stateful load balancing on P4 switches. SilkRoad implements two tables for stateful processing. One table maps virtual IP addresses of services to server instances, another table records active connections identified by hashes of 5-tuples to forward subsequent flows. It applies a Bloom filter to identify new connection attempts and to record those requests in registers to remember client requests that arrive while the pool of server instances changes. In [329], the accompanying demo is described.

HULA [287] implements a link load-based distance vector routing mechanism. Switches in HULA do not maintain the state for every path but the next hops. They send out probes to gather link utilization information. Probe packets are distributed throughout the network on node-specific multicast trees. The probes have a header that contains a destination field and the currently best path utilization to that destination. When a node receives a probe, it updates the best path utilization if necessary, sends one packet clone upstream back to the origin, and forwards copies along the multicast tree further downstream. This way the origin will receive multiple probe packets with different path utilization to a specific destination. Then, flowlets are forwarded onto the best currently available path to its destination.

MP-HULA [288] extends HULA by using load information for n best next hops and compatibility with multipath TCP (MP-TCP). It tracks subflows of

MP-TCP with individual flowlets per sub-flow. MP-HULA aims at distributing those subflows on different paths to aggregate bandwidth. To that end, it is necessary to keep track of the best n next-hops which is done with additional registers and forwarding rules.

Chiang et al. [289] propose a cost-effective congestion-aware load balancing scheme (CCLB). In contrast to HULA, CCLB replaces only the leaf switches with programmable switches, and thus is more cost-effective. They leverage Explicit Congestion Notification (ECN) information in probe packets to recognize congestion in the network and to adapt the load balancing. CCLB further uses flowlet forwarding and is implemented for the bmv2.

W-ECMP [290] is an ECMP-based load balancing mechanism for data centers implemented for P4 switches. Weighted probabilities based on path utilization, are used to randomly choose the best path to avoid congestion. A local agent on each switch computes link utilization for the ports. Regular traffic carries an additional custom packet header that keeps track of the current maximum link utilization on a path. Based on the maximum link utilization, the switches update port weights if necessary.

DASH [291] is an adaptive weighted traffic splitting mechanism that works entirely in the data plane. In contrast to popular weighted traffic splitting strategies such as WCMP, DASH does not require multiple hash table entries. DASH splits traffic based on link weights by portioning the hash space into unique regions.

Pizzutti et al. [292, 293] implement congestion-aware load balancing for flowlets on P4 switches. Flowlets are bursts of packets that are separated by a time gap, e.g., as caused by factors such as TCP dynamics, buffer availability, or link congestion. For distributing subflows on different paths, the congestion state of the last route is stored in a register.

LBAS [294] implements a load balancer to minimize the processing latency at both load balancers and application servers. LBAS does not only reduce the processing latency at load balancers but also takes the application servers' state into account. It is implemented for the Tofino and its average response time is evaluated.

DPRO [295] combines INT with traffic engineering (TE) and reinforcement learning (RL). Network statistics, such as link utilization and switch load, are gathered using an adapted INT approach. An RL-agent inside the controller adapts the link weights based on the minimization of a max-link-utilization objective.

Kawaguchi et al. [296] implement Unsplittable flow Edge Load factor Balancing (UELB). A controller application monitors the link utilization and computes new optimal paths upon congestion. The path computation is based on the UELB problem. The forwarding is implemented in P4 for the bmv2.

AppSwitch [297] implements a load balancer for key-value storage systems. However, the focus lies on a local agent and the control plane communication with the storage server.

Beamer [298] operates in data centers and prevents interruption of connections when they are load-balanced to a different server. To that end, the Beamer controller instructs the new target server to forward packets of the load-balanced connection to the old target server until the migration phase is over.

10.3. Congestion Notification

P4QCN [300] proposes a congestion feedback mechanism where network nodes check the egress ports for congestion before forwarding packets. If a node detects congestion, it calculates a feedback value that is propagated upstream. The mechanism clones the packet that caused the congestion, updates the feedback value in the header, changes the origin of the flow, and forwards it as a feedback packet to the sender. The sender adjusts its sending rate to reduce congestion downstream. The authors describe an implementation where bmv2 is extended by P4 externs for floating-point calculations.

Jiang et al. [301] introduce a novel adjusting advertised windows (AWW) mechanism for TCP. The authors argue that the current calculation of the advertised window in the TCP header is inaccurate because the source node does not know the actual capacity of the network. AWW dynamically updates the advertised window of ACK packets to feedback the network capacity indirectly to the source nodes. Each P4 switch calculates the new AWW value and writes it into the packet header.

EECN [302] presents an enhanced ECN mechanism which piggybacks congestion information if the switch notices congestion. To that end, the ECN-Echo bit is set for traversing ACKs as soon as congestion occurs for a given flow. This enables fast congestion notification without the need for additional control traffic.

Chen et al. [303] present QoSTCP, a TCP version with adapted congestion window growth that enables rate limiting. QoSTCP is based on a marking approach similar to ECN. When a flow exceeds a certain rate, the packet gets marked with a so-called Rate-Limiting Notification (RLN) and the congestion window growth is adapted proportional to the RLN-marked packet rate. Metering and marking is done using P4.

Laraba et al. [304] detect ECN misbehavior with the help of P4 switches. They model ECN as extended finite state machine (EFSM) and store states and variables in registers. If end hosts do not conform to the specified ECN state machine, packets are either dropped or, if possible, the misbehavior is corrected.

10.4. Traffic Scheduling

Sharma et al. [305] introduce a mechanism for per flow fairness scheduling in P4. The concept is based on round-robin scheduling where each flow may send a certain number of bytes in each round. The switch assigns a round number for each arriving packet that depends on the number of sent bytes of flow in the past.

Cascone et al. [306] introduce bandwidth sharing based on sending rates between TCP senders. P4 switches use statistical byte counters to store the sending rate of each user. Depending on the recorded sending rate of the user, arriving packets are pushed into different priority queues.

Bhat et al. [307] leverage P4 switches to translate application layer header information into link-layer headers for better QoS routing. They use Q-in-Q tunneling at the edge to forward packets to the core network and present a bmv2 implementation for HTTP/2 applications, as HTTP/2 explicitly defines a Stream ID that can directly be translated in Q-in-Q tags.

Kfoury et al. [308] present a method to support dynamic TCP pacing with the aid of network state information. A P4 switch monitors the number of active TCP flows, i.e., they monitor the SYN, SYN-ACK, and ACK flags and notify senders about the current network state if a new flow starts or another terminates. To that end, they introduce a new header and show by simulations that the overall throughput increases.

Chen et al. [309] present a design for bandwidth management for QoS with SDN and P4-programmable switches. Their design classifies packets based on a two-rate three-color marker and assigns corresponding priorities to guarantee certain per flow bandwidth. To that end, they leverage the priority queuing capabilities of P4-switches based on the assigned color. Guaranteed traffic goes to a high-priority queue, best-effort traffic goes to a low-priority queue, and traffic that exceeds its bandwidth is simply dropped.

Lee et al. [310] implement a multi-color marker for bandwidth guarantees in virtual networks. Their objective is to isolate bandwidth consumption of virtual networks and provide QoS for its serving flows.

10.5. Traffic Aggregation

Wang et al. [311] introduce aggregation and dis-aggregation capabilities for P4 switches. To reduce the header overhead in the network, multiple small packets are thereby aggregated to a single packet. They leverage multiple register arrays to store incoming small packets in 32 bit chunks. If enough small packets are stored, a larger packet gets assembled with the aid of multiple recirculations; each recirculation step appends a small packet to the aggregated large packet.

RL-SP-DRR [312] is a combination of strict priority scheduling with rate limitation (RL-SP) and deficit round-robin (DRR). RL-SP ensures prioritization of high-priority traffic while DRR enables fair scheduling among different priority classes. They extend bmv2 to support RL-SP-DRR and evaluate it against strict priority queuing and no active queuing mechanism.

10.6. Active Queue Management (AQM)

Turkovic et al. [313] develop an active queue management (AQM) mechanism for programmable data planes. The switches are programmed to collect metadata associated with packet processing, e.g., queue size and load, that are used to prevent, detect, and dissolve congestion by forwarding affected flows on an alternate path. Two possible mechanisms for rerouting in P4 are described. In the first mechanism, primary and backup entries are installed in the forwarding tables and according to the gathered metadata, the suitable action is selected. The second mechanism leverages a local controller on each switch that monitors flows and installs updated forwarding rules when congestion is noticed.

P4-CoDel [314] implements the CoDel AQM mechanism specified in RFC 8289 [330]. CoDel leverages a target and an interval parameter. As long as the queuing delay is shorter than the target parameter, no packets are dropped. If the queuing delay exceeds the target by a value that is at least as large as the interval, a packet is dropped, and the interval parameter is decreased. This procedure is repeated until the queuing delay is under the target threshold again. The interval is then reset to the initial value. To avoid P4 externs, the authors use approximated calculations for floating-point operations.

P4-ABC [316] implements activity-based congestion management (ABC) for P4. ABC is a domain concept where edge nodes measure the activity, i.e., the sending rate, of each user and annotate the value in the packet header. Core nodes measure the average activity of all packets. Depending on the current queue status, the average activity, and activity value in the packet header, a drop decision is made for each packet to prevent congestion. The $P4_{16}$ implementation for the bmv2 requires externs for floating-point calculations.

P4air [317] attempts to provide more fairness for TCP flows with different congestion control algorithms. To that end, P4air groups flows into different categories based on their congestion control algorithm, e.g., loss-, delay- and loss-delay-based. Afterwards, the most aggressive flows are punished based on the previous categorization with packet drops, delay increase, or adjusted receive windows. P4air leverages switch metrics and flow reactions, such as queuing delay and sending rate, to determine the congestion control algorithm used by the flows.

Fernandes et al. [318] propose a bandwidth throttling solution in P4. Incoming packets are dropped with a certain probability depending on the incoming rate of the flow and the defined maximum bandwidth. Rates are measured using time windows and byte counters. Fernandes et al. extend the bmv2 for this purpose.

Wang et al. [319] present an AQM mechanism for video streaming. Data packets are classified as base packets (basic image information) or enhancement packets (additional information to improve the image quality). When the queue size exceeds a certain threshold, enhancement packets are preferably dropped.

SP-PIFO [320] features an approximation of Push-In First-Out (PIFO) queues which enables programmable packet scheduling at line rate. SP-PIFO dynamically adapts the mapping between packet ranks and available strict-priority queues.

Kunze et al. [321] analyze the design of three popular AQM algorithms (RED, CoDel, PIE). They implement PIE in three different variants for Tofino-based P4 hardware targets and show that implementation trade-offs have significant performance impacts.

Harkous et al. [323] use virtual queues implemented in P4 for traffic management. A traffic classifier in the form of MATs assigns a data plane slice identifier to traffic flows. P4 registers are used to implement virtual queues for each data plane slice for traffic management.

10.7. Traffic Offloading

Andrus et al. [324] propose to offload video stream processing of surveillance cameras to P4 switches. The authors propose to offload stream processing for storage to P4 switches. In case the analytics software detected an event, it enables a multistage pipeline on the P4 switch. In the first step, video stream data is replicated. One stream is further sent to the analytics software, the other stream is dedicated to the video storage. The P4 switch filters out control packets and rewrites the destination IP address of all video packets to the video storage.

Ibanez et al. [325] try to tackle the problem of P4's packet-by-packet programming model. Many tasks, such as periodic updates, require either hardware-specific capabilities or control-plane interaction. Processing capabilities are limited to enqueue events, i.e., data plane actions are only triggered if packets arrive. To eliminate this problem, the authors propose a new mechanism for event processing using the P4 language.

Kfoury et al. [326] propose to offload media traffic to P4 switches which act as relay servers. A SIP server receives the connection request, replaces IP and port information with the relay server IP and port, and forwards the request to the receiver. Afterwards, the media traffic is routed through the relay server.

Falcon [327] offloads task scheduling to programmable switches. Job requests are sent to the switch and the switch assigns a task in first-come-first-serve order to the next executor in a pool of computation nodes. Falcon reduces the scheduling overhead by a factor of 26 and increase scheduling throughput by a factor of 25 compared to state-of-the-art schedulers.

Osinski et al. [328] present vBNG, a virtual Broadband Network Gateway (BNG). Some components, such as PPPoE session handling, are offloaded to programmable switches.

10.8. Summary and Analysis

The research domain of traffic management and congestion control benefits from three core properties of P4: *custom packet headers*, *flexible header processing* and *target-specific packet header processing functions.* Data center switching mainly relies on packet header parsing of well-known protocols, such as IPv4/v6 or MPLS. More advanced protocol solutions, such as VXLAN and BIER, can be implemented by leveraging the *flexible packet header processing* property of P4. The presented efforts on load balancing (Section 10.2) also use this property of P4 to implement novel approaches. *Target-specific packet header processing functions* such as externs are widely used in Section 10.3. Most works leverage externs such as metering and marking which may not be supported on all hardware targets. A similar phenomenon appears in Section 10.4. Here, many papers are based on priority queues. The approaches on AQM in Section 10.6 encounter similar limitations. Floating-point operations are not part of the P4 core. Some targets may provide an extern for this functionality. Multiple works avoid this problem by either using approximations or by relying on self-defined externs in software.

11. Applied Research Domains: Routing and Forwarding

We describe applied research on source routing, multicast, publish-subscribe-systems, named data networking, data plane resilience, and other fields of application. Table 7 shows an overview of all the work described. At the end of the section, we summarize the work and analyze it with regard to P4's core features described in Section 8.1.

11.1. Source Routing

With source routing, the source node defines the processing of the packet throughout the network. To that end, a header stack is often added to the packet to specify the operations the other network devices should execute.

Lewis et al. [331] implement a simple source routing mechanism with P4 for the bmv2. The authors introduce a header stack to specify the processing of the packet towards its destination. That header stack is constructed and pushed onto the packet by the source node. Network devices match the header segments to determine how the packet should be processed.

Luo et al. [333] implement segment routing with P4. They introduce a header which contains segments that identify certain operations, e.g., forwarding the packet towards a specific destination or over a specific link, updating header fields, etc. Network nodes process packets according to the topmost segment in the segment routing header and remove it after successful execution.

Kushwaha et al. [335] implement bitstream, a minimalistic programmable data plane for carrier-class networks, in P4 for FPGAs. The focus of bitstream is to provide a programmable data plane while ensuring several carrier-grade properties, like deterministic latencies, short restoration time, and per-service measurements. To that end, the authors implement a source routing approach in P4 which leaves the configuration of the header stack to the control plane.

The authors of [336] show a demo of segment routing over IPv6 data plane (SRv6) implementation in P4. It leverages the novel uSID instruction set for SRv6 to improve scalability and MTU efficiency.

11.2. Multicast

Multicast efficiently distributes one-to-many traffic from the source to all subscribers. Instead of sending individual packets to each destination, multicast packets are distributed in tree-like structures throughout the network.

Bit Index Explicit Replication (BIER) [386] is an efficient transport mechanism for IP multicast traffic. In contrast to traditional IP multicast, it prevents subscriber-dependent forwarding entries in the core network by leveraging a BIER header that contains all destinations of the BIER packet. To that end, the BIER header contains a bit string where each bit corresponds to a specific destination. If a destination should receive a copy of the BIER packet, its corresponding bit is activated in the bit string in BIER header of the packet. Braun et al. [337] present a demo implementation of BIER-based multicast in P4. Merling et al. [339] implement BIER-based multicast with fast reroute capabilities in P4 for the bmv2 and for the Tofino [340].

Table 7: Overview of applied research on routing and forwarding (Section 11).

Research work	Year	Targets	Code
Source Routing (11.1)			
Lewis et al. [331]	2018	bmv2	[332]
Luo et al. [333]	2019	bmv2	[334]
Kushwaha et al. [335]	2020	Xilinx Virtex-7	
Abdelsalam et al. [336]	2020	bmv2	
Multicast (11.2)			
Braun et al. [337]	2017	bmv2	[338]
Merling et al. [339, 340]	2020/21	bmv2, Tofino	[341, 342]
Elmo [343]	2019	-	[344]
PAM [345]	2020	bmv2	
Publish/Subscribe Systems (11.3)			
Wernecke et al. [346, 347, 348, 349]	2018/19	bmv2	
Jepsen et al. [350]	2018	Tofino	
Kundel et al. [351]	2020	bmv2	[352]
FastReact-PS [353]	2020	-	
Named Data Networks (11.4)			
NDN.p4 [354, 355]	2016/18	bmv2	[356, 357]
ENDN [358]	2020	bmv2	
Data Plane Resilience (11.5)			
Sedar et al. [359]	2018	bmv2	[360]
Giesen et al. [361]	2018	Tofino, Xilinx SDNet	
SQR [362]	2019	bmv2, Tofino	[363]
P4-Protect [364]	2020	bmv2, Tofino	[365, 366]
Hirata et al. [367]	2019	-	
Lindner et al. [368]	2020	bmv2, Tofino	[369, 370]
D2R [371]	2019	bmv2	
PURR [372]	2019	bmv2, Tofino	
Blink [373]	2019	bmv2, Tofino	[374]

Research work	**Year**	**Targets**	**Code**
Other Fields of Applications (11.6)			
Contra [375]	2019	-	
Michel et al. [376]	2016	bmv2	
Baktir et al. [377]	2018	bmv2	
Froes et al. [378]	2020	bmv2	
QROUTE [379]	2020	bmv2	
Gimenez et al. [380]	2020	bmv2	
Feng et al. [381]	2019	bmv2	
PFCA [382]	2020	bmv2	
McAuley et al. [383]	2019	bmv2	
R2P2 [384]	2019	Tofino	[385]

Elmo [343] is a system for scalable multicast in multi-tenant datacenters. Traditional IP multicast maintains subscriber dependent state in core devices to forward multicast traffic. This limits scalability, since the state in the core network has to be updated every time subscribers change. Elmo increases scalability of IP multicast by moving a certain subscriber-dependent state from the core devices to the packet header.

Priority-based adaptive multicast (PAM) [345] is a control protocol for data center multicast which is implemented by the authors in P4. Network administrators define different policies regarding priority, latency, completion time, etc., which are installed on the core switches. The network devices than monitor link loads and adjust their forwarding to fulfill the policies.

11.3. Publish/Subscribe Systems

Publish/subscribe systems are used for data distribution. Subscribers are able to subscribe to announced topics. Based on the subscriptions, the data packets are distributed from the source to all subscribers.

Wernecke et al. [346, 347, 348, 349] implement a content-based publish/subscribe mechanism with P4. The distribution tree to all subscribers is encoded directly in the header of the data packets. To that end, the authors introduce a header stack which is pushed onto the packet by the source. Each element in the stack consists of an ID and a value. When a node receives a packet, it checks whether the header stack contains an element with its own ID. If so, the value determines to which neighbors the packet has to be forwarded.

Jepsen et al. [350] introduce a description language to implement publish/subscriber systems. The data plane description is translated into a static pipeline and dynamic filters. The static pipeline is a P4 program that describes a packet processing pipeline for P4 switches, the dynamic filters are the forwarding rules of the match-action tables that may change during operation, e.g., when subscriptions change.

Kundel et al. [351] propose two approaches for attribute/value encoding in packet headers for P4-based publish/subscribe systems. This reduces the header overhead and facilitates adding new attributes which can be used for subscription by hosts.

FastReact-PS [353] is a P4-based framework for event-based publish/subscribe in industrial IoT networks. It supports stateful and stateless processing of complex events entirely in the data plane. Thereby, the forwarding logic can be dynamically adjusted by the control plane without the need for recompilation.

11.4. Named Data Networking

Named data networking (NDN) is a content-centric paradigm where information is requested with resource identifiers instead of destinations, e.g., IP addresses. Network devices cache recently requested resources. If a requested resource is not available, network devices forward the request to other nodes.

NDN.p4 [354] implements NDN without caching for P4. However, the implementation cannot cache requests because of P4-related limitations with stateful storage. Miguel et al. [355] leverage the new functionalities of $P4_{16}$ to extend NDN.p4 by a caching mechanism for requests and optimize its operation. The caching mechanism is implemented with P4 externs.

Enhanced NDN (ENDN) [358] is an advanced NDN architecture. It offers a larger catalog of content delivery features like adaptive forwarding, customized monitoring, in-network caching control, and publish/subscribe forwarding.

11.5. Data Plane Resilience

Sedar et al. [359] implement a fast failover mechanism without control plane interaction for P4 switches. The mechanism uses P4 registers or metadata fields for bit strings that indicate if a particular port is considered up or down. In a match-action table, the port bit string provides an additional match field to determine whether a particular port is up or down. Depending on the port status, default or backup actions are executed. The authors rely on a local P4 agent to populate the port bit strings.

Giesen et al. [361] introduce a forward error correction (FEC) mechanism for P4. Commonly, unreliable but not completely broken links are avoided. As this happens at the cost of throughput, the proposed FEC mechanism facilitates the usage of unreliable links. The concept features a link monitoring agent that polls ports to detect unreliable connections. When a packet should be forwarded over such a port, the P4 switch calculates a resilient encoding for the packet which is then decoded by the receiving P4 switch.

Shared Queue Ring (SQR) [362] introduces an in-network packet loss recovery mechanism for link failures. SQR caches recent traffic inside a queue with slow processing speed. If a link failure is detected, the cached packets can be sent over an alternative path. While P4 does not offer the possibility to store packets for a certain amount of time, the authors leverage the cloning operation of P4 to keep packets inside the buffer. If a cached packet has not yet met its delay, it gets cloned to another egress port which takes some time. This procedure is repeated until the packet has been stored for a given time span.

P4-Protect [364] implements 1+1 protection for IP networks. Incoming packets are equipped with a sequence number, duplicated, and sent over two disjoint paths. At an egress point, the first version of each packet is accepted and forwarded. As a result, a failure of a single path can be compensated without additional signaling or reconfiguration. P4-Protect is implemented for the bmv2 and the Tofino. Evaluations show that line-rate processing with 100 Gbit/s can be achieved with P4-Protect at the Tofino.

Hirata et al. [367] implement a data plane resilience scheme based on multiple routing configurations. Multiple routing configurations with disjoint paths are deployed, and a header field identifies the routing configuration according to which packets are forwarded. In the event of a failure, a routing configuration is chosen that avoids the failure.

Lindner et al. [368] present a novel prototype for in-network source protection in P4. A P4-capable switch receives sensor data from a primary and secondary sensor, but forwards only the data from the primary sensor if available. It detects the failure of the primary sensor and then transparently forwards data from a secondary sensor to the application. Two different mechanisms are presented. The *counter-based* approach stores the number of packets received from the secondary sensor since the last packet from the primary sensor has been received. The *timer-based* approach stores the time of the last arrival of a packet from the primary sensor and considers the time since then. If certain thresholds are exceeded, the P4-switch forwards the data from the secondary sensor.

D2R [371] is a data-plane-only resilience mechanism. Upon a link failure, the data plane calculates a new path to the destination using algorithms like breadth-first search and iterative deepening depth-first search. As one pipeline iteration has not enough processing stages to compute the path, recirculation is leveraged. In addition, *Failure Carrying Packets (FCP)* is used to propagate the link failure inside the network. While the authors claim that their architecture works with hardware switches, e.g., the Tofino, they only present and evaluate a bmv2 implementation.

Chiesa et al. [372] propose a primitive for reconfigurable fast ReRoute (PURR) which is a FRR primitive for programmable data planes, in particular for P4. For each destination, suitable egress ports are stored in bit strings. During packet processing, the first working suitable egress port is determined by a set of forwarding rules. Encoding based on *Shortest Common Supersequence* guarantees that only few additional forwarding rules are required.

Blink [373] detects failures without controller interaction by analyzing TCP signals. The core concept is that the behavior of a TCP flow is predictable when it is disrupted, i.e., the same packet is retransmitted multiple times. When this information is aggregated over multiple flows, it creates a characteristic failure signal that is leveraged by data plane switches to trigger packet rerouting to another neighbor.

Contra [375] introduces performance-aware routing with P4. Network paths are ranked according to policies that are defined by administrators. Contra applies those policies and topology information to generate P4 programs that define the behavior of forwarding devices. During runtime, probe packets are used to determine the current network state and update forwarding entries for best compliance with the defined policies.

Michel et al. [376] introduce identifier-based routing with P4. The authors argue that IP addresses are not fine-granular enough to enable adequate forwarding, e.g., in terms of security policies. The authors introduce a new header that contains an identifier token. Before sending packets, applications transmit information on the process and user to a controller that returns an identifier that is inserted into the packet header. P4 switches are programmed to forward packets based on that identifier.

Baktir et al. [377] propose a service-centric forwarding mechanism for P4. Instead of addressing locations, e.g., by IP addresses, the authors propose to use location-independent service identifiers. Network hosts write the identifier of the desired service into the appropriate header field, the switches then make forwarding decisions based on the identifier in the packet header. With this approach, the location of the service becomes less important since the controller simply updates the forwarding rules when a service is migrated or load balancing is desired.

Froes et al. [378] classify different traffic classes which are identified by a label. Packet forwarding is based on that controller-generated label instead of IP addresses. The traffic classes have different QoS properties, i.e., prioritization of specific classes is possible. To that end, switches leverage multiple queues to process traffic of different traffic classes.

QROUTE [379] is a quality of service (QoS) oriented forwarding scheme in P4. Network devices monitor their links and annotate values, e.g., jitter or delay, in the packet header so that downstream nodes can update their statistics. Furthermore, packet headers contain constraints like maximum jitter or delay. According to those values, forwarding decisions are made by the network devices.

Gimenez et al. [380] implement the recursive internet-work architecture (RINA) in P4 for the bmv2. RINA is a networking architecture which sees computer networking as a type of inter-process communication where layering should be based on scope/scale instead of function. In general, efficient implementations require hardware support. However, up to date only software-based implementations are available. The authors hope that with the advance of programmable hardware in the form of P4, hardware-based RINA will soon be possible.

Feng et al. [381] implement information-centric network (ICN) based forwarding for HTTP. To that end, they propose mechanisms to convert packets from ICN to HTTP packets and vice-versa.

PFCA [382] implements a forwarding information base (FIB) caching architecture in the data plane. To that end, the P4 program contains multiple

MATs that are mapped to different memory, i.e., TCAM, SRAM, dynamic random access memory (DRAM), with different properties regarding lookup speed. Counters keep track of cache hits to move (un)popular rules to other tables.

McAuley et al. [383] present a hybrid error control booster (HEC) that can be deployed in wireless, mobile, or hostile networks that are prone to link or transport layer failures. HECs increase the reliability by applying a modified Reed-Solomon code that adds parity packets or additional packet block acknowledgments. P4 targets include an error control processor that implements this functionality. It is integrated into the P4 program as P4 extern so that the data plane can exchange HEC packets with it. A remote control plane includes the booster manager that controls HEC operations and parameters on the P4 targets via a data plane API.

R2P2 [384] is a transport protocol based on UDP for latency-critical RPCs optimized for datacenters or other distributed infrastructure. A router module implemented in P4 or DPDK is used to relay requests to suitable servers and perform load balancing. It may also perform queuing if no suitable server is available. The goal of R2P2 is to overcome problems that typically come with TCP-based RPC systems, e.g., problems with load distribution and head-of-line-blocking.

11.7. Summary and Analysis

The research domain of routing and forwarding greatly benefits from P4's core features. First, the *definition and usage of custom packet headers* enables administrators to tailor the packet header to the specific use case. Two examples are source routing (Section 11.1) and multicast (Section 11.2). Both areas leverage custom headers to define lightweight mechanisms based on additional information in the packet header which are not part of any standard protocol. Although most of the projects were developed only for the bmv2, they should be easily portable to hardware platforms as more complex, target specific operations are not required. Second, users are able to define *flexible packet header processing* depending on the information in the packet header, e.g., publish/-subscribe systems (Section 11.3), named data networks (Section 11.4), and data plane resilience (Section 11.5). Parametrized custom actions and (conditional) application of multiple MATs allow for adaptable packet processing for many specific use cases. Similar to the previous P4 core feature, most projects were developed for the bmv2 but they should be easy to transfer if no target-specific actions are used. Third, we found that many papers in the area of data plane resilience (Section 11.5) leverage *target-specific packet header processing functions.* Often registers are used to store information whether egress ports are up or down to execute backup actions if necessary. Most projects were implemented for the hardware platform Tofino. As a result, the implementations are highly target-specific and transferring them to other hardware platforms highly depends on the capabilities of the target platform and the used externs.

12. Applied Research Domains: Advanced Networking

We describe applied research on cellular networks (4G/5G), Internet of things (IoT), industrial networking, Time-Sensitive Networking (TSN), network function virtualization (NFV), and service function chains (SFCs). Table 8 shows an overview of all the work described. At the end of the section, we summarize the work and analyze it with regard to P4's core features described in Section 8.1.

12.1. Cellular Networks (4G/5G)

P4EC [387] builds a local exit for LTE deployments with cloud-based EPC services. A programmable switch distinguishes traffic and reroutes traffic for edge computing. Non-critical traffic is forwarded to the cloud-based EPC.

The Trellis switch fabric (introduced in Section 10.1) features the spgw.p4 profile [281, 277], an implementation of a Serving and PDN Gateway (SPGW) for 5G networking. ONOS runs an SPGW-u application that implements the 3GPP control and user plane separation (CUPS) protocol to create, modify, and delete GPRS tunneling protocol (GTP) sessions. It provides support for GTP en- and decapsulation, filtering, and charging.

SMARTHO [389] proposes a handover framework for 5G. Distributed units (DUs) include real-time functions for multiple 5G radio stations. Several DUs are controlled by a central unit (CU) that includes non-real-time control functions. P4 switches are part of the CU and all DU nodes. SMARTHO introduces a P4-based mechanism for preparing handover sequences for user devices that take a fixed path among 5G radio stations controlled by DUs. This decreases the overall handover time, e.g., for users traveling in a train.

Aghdai et al. [390] propose a P4-based transparent edge gateway (EGW) for mobile edge computing (MEC) in LTE or 5G networks. Delay-sensitive and bandwidth-intense applications need to be moved from data centers in the core network to the edge of the radio access network (RAN). 5G networks rely on GTP-U for encapsulating IP packets from the mobile user to the core network. IP routers in between forward packets based on the outer IP address of GTP-U frames. The authors deploy EGWs as P4 switches at the edge of the IP transport network where service operators can deploy scalable network functions or services. Each MEC service gets a virtual IP address, the P4-based EGWs parse the inner IP destination address of GTP-U. If it sees traffic targeting a virtual IP address of a MEC service, it forwards it to the IP address of one of the serving instances of the MEC application. In their follow-up work [391], the authors extend EGWs by a handover mechanism for migrating network state.

GRED [392] is an efficient data placement and retrieval service for edge computing. It tries to improve routing path lengths and forwarding table sizes. They follow a greedy forwarding approach based on DT graphs, where the forwarding table size is independent of the network size and the number of flows in the network. GRED is implemented in P4, but the authors do not specify on which target.

Table 8: Overview of applied research on advanced networking (Section 12).

Research work	**Year**	**Targets**	**Code**
Cellular Networks (4G/5G) (12.1)			
P4EC [387]	2020	Tofino	
Trellis [281]	-	-	[388]
SMARTHO [389]	2018	bmv2	
Aghdai et al. [390, 391]	2018/19	Netronome	
GRED [392]	2019	bmv2	
HDS [393]	2020	-	
Shen et al. [394]	2019	Xilinx SDNet	
Lee et al. [395]	2019	Tofino	
Ricart-Sanchez et al. [396]	2019	NetFPGA-SUME	
Singh et al. [397]	2019	Tofino	
TurboEPC [398]	2020	Netronome	
Vörös et al. [399]	20200	Tofino	
Lin et al. [400]	2019	Tofino	
Internet of Things (12.2)			
BLESS [401]	2017	PISCES	
Muppet [402]	2018	PISCES	
Wang et al. [403]	2019	Tofino	
Madureira et al. [404]	2020	bmv2	
Engelhard et al. [405]	2019	bmv2	
Industrial Networking (12.3)			
FastReact [406]	2018	bmv2	
Cesen et al. [407]	2020	bmv2	
Kunze et al. [408]	2020	Tofino, Netronome	
Time-Sensitive Networking (TSN) (12.4)			
Rüth et al. [409]	2018	Netronome	
Kannan et al. [410]	2019	Tofino	
Kundel et al. [411]	2019	Tofino	

Research work	Year	Targets	Code
Network Function Virtualization (NFV) (12.5)			
Kathará [412]	2018	-	
P4NFV [413]	2018	bmv2	
Osiński et al. [414]	2019	-	
Moro et al. [415]	2020	-	
DPPx [416]	2020	bmv2	
Mohammadkhan et al. [417]	2019	Netronome	
FOP4 [418, 419]	2019	bmv2, eBPF	
PlaFFE [420]	2020	Netronome	
Service Function Chains (SFCs) (12.6)			
P4SC [421, 422]	2019	bmv2, Tofino	[423]
Re-SFC [424]	2019	bmv2	
FlexMesh [425]	2020	bmv2	
P4-SFC [426]	2019	bmv2, Tofino	[427]

HDS [393] is a low-latency, hybrid, data sharing framework for hierarchical mobile edge computing. The data location service is divided into two parts: intra-region and inter-region. The authors present a data sharing protocol called Cuckoo Summary for fast data localization for the intra-region part. Further, they developed a geographic routing scheme to achieve efficient data location with only one overlay hop in the inter-region part.

Shen et al. [394] present an FGPA-based GTP engine for mobile edge computing in 5G networks. Communication between the 5G back-haul and the conventional Ethernet requires de- and encapsulation of traffic with GTP. As most network entities do not have the capability to process GTP, the authors leverage P4-programmable hardware for this purpose.

Lee et al. [395] evaluate the performance of GTP-U and SRv6 stateless translation as GPT-U cannot be replaced by SRv6 without a transition period. To that end, they implement GTP and SRv6 on P4-programmable hardware. They found that there are no performance drops if stateless translation is used and that SRv6 stateless translation is acceptable for the 5G user plane.

Ricart-Sanchez et al. [396] propose an extension for the P4-NetFPGA framework for network slicing between different 5G users. The authors extend the capabilities of the P4 pipeline and implement their mechanism on the NetFPGA-SUME. However, the authors do not provide any details about their implementation.

Singh et al. [397] present an implementation for the Evolved Packet Gateway (EPG) in the Mobile Packet Core of 5G. They show that they can offload the functionality to programmable switching ASICs and achieve line rate with low latency and jitter while scaling up to 1.7 million active users.

TurboEPC [398] presents a redesign of the mobile packet core where parts of the control plane state is offloaded to programmable switches. State is stored in MATs. The switches then process a subset of signaling messages within the data plane itself, which leads to higher throughput and reduced latency.

Vörös et al. [399] propose a hybrid approach for the next generation NodeB (gNB) where the majority of packet processing is done by a high-speed P4-programmable switch. Additional functions, such as ARQ or ciphering, are offloaded to external services such as DPDK implementations.

Lin et al. [400] enhance the Content Permutation Algorithm (eCPA) for secret permutation in 5G. Packet payloads are split into code words and shuffled according to a secret cipher. They implement eCPA for switches of the Inventec D5264 series.

12.2. Internet of Things (IoT)

BLESS [401] implements a Bluetooth low energy (BLE) service switch based on P4 that acts as a proxy enabling flexible, policy-based switching and in-network operations of IoT devices. BLE devices are strictly bound to a central device such as a smartphone or tablet. IoT usage requires cloud-based solutions where central devices connect to an IoT infrastructure. The authors propose a BLE service switch (BLESS) that is transparently inserted between peripheral and central devices and acts like a transparent proxy breaking up the peer-to-peer model. It maintains BLE link layer connections to peripheral devices within its range. A central controller implements functionalities such as service discovery, access policy enforcement, and subscription management so that features like service slicing, enrichment, and composition can be realized by BLESS.

Muppet [402] extends BLESS by supporting the Zigbee protocol in parallel to BLE. In addition to the features of BLESS, inter-protocol services between Zigbee and BLE and BLE/Zigbee and IP protocols are introduced. An example for the latter are HTTP transactions that are automatically sent out by the switch if it sees a specified set of BLE/Zigbee transactions. The data plane implementation of BLESS is extended by protocol-dependent packet parsers and processing and support for encrypted Zigbee packets via packet recirculation.

Wang et al. [403] implement aggregation and disaggregation of small IoT packets on P4 switches. For a small IoT packet, the header holds a large proportion of the packet's total size. In large streams of IoT packets, this causes high overhead. The current aggregation techniques for IoT packets are implemented by external servers or on the control plane of switches, both resulting in low throughput and added latency. Therefore, the authors propose an implementation directly on P4 switches where IoT packets are buffered, aggregated, and encapsulated in UDP packets with a custom flag-header, type, and padding. In disaggregation, the incoming packet is cloned to stripe out the single messages until all messages are separated.

Madureira et al. [404] present the *Internet of Things Protocol (IoTP)*, an L2 communication protocol for IoT data planes. The main purpose of IoTP is data aggregation at the network level. IoTP introduces a new, fixed header and is compatible with any forwarding mechanism. The authors implemented IoTP for the bmv2 and store single packets of a flow in registers until the data can be aggregated.

Engelhard et al. [405] present a system for massive wireless sensor networks. They implement a physically distributed, and logically centralized wireless access systems to reduce the impairment by collisions. P4 is leveraged as connection between a physical access point and a virtual access point. To that end, they extend the bmv2 to provide additional functionality. However, they give information about their P4 program only in form of a decision flow graph.

12.3. Industrial Networking

FastReact [406] outsources sensor data packet processing from centralized controllers to P4 switches. The sensor data is recorded in variable-length time series data stores where an additional field holds the current moving average calculated on the time series. Both data for all sensors can be polled by a central controller. For controlling actuators directly on the data plane, FastReact supports the formulation of control logic in conjunctive normal form (CNF). It is mapped to actions to either forward signal data to the controller, discard it, or directly send it to the actuator. FastReact also features failure recovery directly on the switch. For every sensor and actuator, timestamps for the last received packets along a timeout limit is recorded. If failures are detected, sensor data are forwarded following failover rules with backup actuators for particular sensors.

Cesen et al. [407] leverage P4-capable switches to move control logic to the network. Control applications reside in controllers that are responsible for emergency intervention, e.g., if a given threshold is exceeded. The connection to the controller may be faulty and, therefore, controller intervention may not be fast enough. In this work, the authors generate emergency packets, i.e., stop commands, directly in the data plane. The action is triggered if the switch receives a packet with a specific payload.

Kunze et al. [408] investigate the applicability of in-network computing to industrial environments. They offload a simple task, i.e., coordinate transformation, to different programmable P4 targets. They come to the conclusion, that, while in general possible, even simple task have heavy demands on programmable network devices and that offloading may lead to inaccurate results.

12.4. Time-Sensitive Networking (TSN)

Rüth et al. [409] introduce a scheme for implementing in-network control mechanisms for linear quadratic regulators (LQR). LQRs can be described by a multiplication of a matrix and a vector. The vector describes the control of the actuator, the matrix describes the current system state. The result of the multiplication is a control command. The destination of a switch describes a specific actuator. When a switch receives a control packet, it matches the destination of the packet onto a match-and-action table. The lookup provides the control vector for the actuator. The control vector from the lookup is then multiplied with the system state matrix that is stored in a register to calculate the control command for the actuator. The resulting control command is written into the packet header and the packet is forwarded to the target actuator.

Kannan et al. [410] introduce the Data Plane Time synchronization Protocol (DPTP) for distributed applications with computations directly on the P4 data plane. DPTP follows a request-response model, i.e., all P4 switches request the global time from a designated master switch. Therefore, each switch features a local control plane that generates time requests sent to the master switch. Additionally, the control plane handles overflows in time calculation for administration.

Kundel et al. [411] demonstrate timestamping with nanosecond accuracy. They describe a simple setup with a Tofino-based switch and a breakout cable to connect two ports of the switch. In the experiment, timestamps at the moment of sending and reception are recorded in the packet header. The authors compare those two timestamps to show that very fine-grained measurements are possible.

12.5. Network Function Virtualization (NFV)

Kathará [412] runs NFs as P4 programs either on software or hardware targets. For software-based deployment, the framework leverages Docker containers that run NFs as container images or individual setups for Quagga, Open vSwitch, or bmv2 container images. For hardware-based deployment on P4 switches, NFs are either replicated on every P4 switch or distributed on multiple P4 switches as needed. In both cases, a load balancer or service classifier forwards flows to the appropriate P4 switch. As a main advantage, P4 programs can be shifted between the bmv2-based P4 software targets and hardware targets depending on the required performance.

P4NFV [413] also deals with the idea of running NFs either on software- or hardware-based P4 targets. The authors adopt the ETSI NFV architecture with control and monitoring entities and add a layer that abstracts various types of software- and hardware-based P4 targets as P4 nodes. For optimized deployment, the targets performance characteristics are part of the P4 node description. For runtime reconfiguration, the authors propose two approaches. In pipeline manipulation, the P4 program features multiple match-action pipelines that can be enabled or disabled by setting register flags. In program reload, a new P4 program is compiled and loaded to the P4 target. The authors propose to perform state management and migration either directly on the data plane or via a control plane.

Osiński et al. [414] use P4 to offload the data plane of virtual network functions (VNFs) into a cloud infrastructure by allowing VNFs to inject small P4 programs into P4 devices like SmartNICs or top-of-rack switches. This results in better performance and a microservice-based approach for the data plane. A new P4 architecture model that integrates abstractions used to develop VNF data planes was developed.

Moro et al. [415] present a framework for NF decomposition and deployment. They split NFs into components that can run on CPUs or that can be offloaded to specific programmable hardware, e.g., P4 programmable switches. The presented orchestrator combines multiple functions into a single P4 program that can be deployed to programmable switches.

DPPx [416] implements a framework for P4-based data plane programmability and exposure which allows enhancing NFV services. They introduce data plane modules written in P4 which can be leveraged by the application plane. As an example, a dynamic optimization of packet flow routing (DOPFR) is implemented using DPPx.

Mohammadkhan et al. [417] provide a unified P4 switch abstraction framework where servers with software NFs and P4-capable SmartNICs are seen as one logical entity by the SDN controller. They further leverage Mixed Integer Linear Programming (MILP) to determine partitioning of P4 tables for optimal placement of NFs.

FOP4 [418] [419] implements a rapid prototyping platform that supports container-based, P4-switch-based, and SmartNIC-based NFs. They argue that a prototyping platform is needed to quickly develop and evaluate new NFV use cases.

PlaFFE [420] introduces NFV offloading where some features of VNFs or embedded Network Functions (eNFs) are executed on SmartNICs using P4. Additionally, P4 is used to steer traffic either through the eNFs or through VNFs using SR-IOV.

12.6. Service Function Chains (SFCs)

P4SC [421] [422] implements a SFC framework for P4 targets. SFCs are described as directed acyclic graph of service functions (SFs). In P4SC, SFs are represented by blocks. Each block has a unique identifier, a P4 program for ingress processing, and a P4 program for egress processing. P4SC includes 15 SF blocks, e.g., L2 forwarding, which are extracted from switch.p4. After the user specified all SFCs for a particular P4 target, the P4SC converter merges the directed acyclic graphs of all SFCs with an LCS-based algorithm into an intermediate representation. Then, the P4SC generator creates the final P4 program based on the intermediate representation to be deployed onto the P4 target. P4 program generation includes runtime management, i.e., the generator creates one API per SFC while hiding SF-specific details, e.g., names of particular match-and-action tables.

Re-SFC [424] improves P4SC's resource usage by using resubmit operations. If the specified order of SFs in an SFC does not match the pre-embedded SF of the P4 switch, incoming flows cannot be processed. P4SC solves this problem by permitting redundant NF embeds, i.e., if SFs of one SFC are required by another SFCs, those SFs are just replicated. To reduce the costly usage of match-and-action tables, Re-SFC introduces resubmit actions where packets are re-bounced to the ingress.

FlexMesh [425] tackles the problem of fixed SFC flow control, i.e., when the specified order of SFs does not match the pre-embedded SF, by leveraging MATs. SFs can be dynamically bypassed, and recirculation is used to build any desired SF chain.

P4-SFC [426] is an SFC framework based on MPLS segment routing and NFV. P4 is used to implement a traffic classifier. A central orchestrator deploys service functions as VNFs and configures the traffic classifier based on definitions of SFCs.

12.7. Summary and Analysis

As the research domain of advanced networking covers different topics, almost all core properties of P4 are covered. The area of cellular networks (Section 12.1) greatly benefits from the *definition and usage of custom packet headers* as many works are based on tunneling technologies, such as GTP. Further, *flexible packet header processing* allows implementing new 5G concepts such as gNB or EPG. Some use cases still require offloading tasks to specialized hardware or software by leveraging the *target-specific packet header processing function* property of P4, e.g., for ARQ or ciphering in the context of gNB. Network function virtualization (NFV) (Section 12.5) benefits from *flexible development and deployment* as single network functions (NFs) can be replaced or relocated during operation. New protocols and extensions to existing protocols presented in Section 12.6 rely on *definition and usage of custom packet headers* and *flexible packet header processing.*

13. Applied Research Domains: Network Security

We describe applied research on firewalls, port knocking, DDoS attack mitigation, intrusion detection systems, connection security, and other fields of application. Table 9 shows an overview of all the work described. At the end of the section, we summarize the work and analyze it with regard to P4's core features described in Section 8.1.

13.1. Firewalls

Ricart-Sanchez et al. [428] present a 5G firewall that analyzes GTP data transmitted between edge and core networks. P4 allows an implementation of parsing and matching GTP header fields such as 5G user source IP, 5G user destination IP, and identification number of the GTP tunnel. The P4 pipeline implements an allow-by-default policy, DROP actions for specific sets of keys can be installed via a data plane API. In a follow-up work [429], the authors extend the 5G firewall by support for multi-tenancy with VXLAN.

CoFilter [430] implements an efficient flow identification scheme for stateful firewalls in P4. To solve the problem of limited table sizes on SDN switches, flow identifiers are calculated by applying a hashing function to the 5-tuple of every packet directly on the switch. The proposed concept includes a novel hash rewrite function that is implemented on the data plane. It resolves hash commission and hash table optimization using an external server.

P4Guard [431] replaces software-based firewalls by P4-based virtual firewalls in the VNGuard [479] system. VNGuard introduces controller-based deployment and management of virtual firewalls with the help of SDN and NFV. The

Table 9: Overview of applied research on network security (Section 13).

Research work	Year	Targets	Code
Firewalls (13.1)			
Ricart-Sanchez et al. [428, 429]	2018/19	NetFPGA-SUME	
CoFilter [430]	2018	Tofino	
P4Guard [431]	2018	bmv2	
Vörös and Kiss [432]	2016	p4c-behavioral	
Port Knocking (13.2)			
P4Knocking [433]	2020	bmv2	
Almaini et al. [434]	2019	bmv2	
DDoS Mitigation Mechanisms (13.3)			
LAMP [435]	2018	bmv2	
TDoSD@DP [436, 437]	2018/19	bmv2	
Kuka et al. [438]	2019	Xilinx UltraScale+, Intel Stratix 10	
Paolucci et al. [439, 440]	2018/19	bmv2, NetFPGA-SUME	
ML-Pushback [441]	2019	-	
Afek et al. [442]	2017	p4c-behavioral	
Cardoso Lapolli et al. [443]	2019	bmv2	[444]
Cai et al. [445]	2020	-	
Lin et al. [446]	2020	bmv2	
Musumeci et al. [447]	2020	bmv2	
DIDA [448]	2020	bmv2	
Dimolianis et al. [449]	2020	Netronome	
Scholz et al. [450]	2020	bmv2, T_4P_4S, Netronome, NetFPGA SUME	[451]
Friday et al. [452]	2020	bmv2	
NetHide [453]	2018	-	
Intrusion Detection Systems & Deep Packet Inspection (13.4)			
P4ID [454]	2019	bmv2	
Kabasele and Sadre [455]	2018	bmv2	
DeepMatch [456]	2020	Netronome	[457]
Qin et al. [458]	2020	bmv2, Netronome	[459]
SPID [460]	2020	bmv2	

Research work	Year	Targets	Code
Other Fields of Application (13.6)			
Chang et al. [461]	2019	bmv2	
Clé [462]	2019	-	
P4DAD [463]	2020	bmv2	
Chen [464]	2020	Tofino	[465]
Gondaliya et al. [466]	2020	NetFPGA SUME	
Poise [467]	2020	Tofino	[468]
Connection Security (13.5)			
P4-MACsec [469]	2020	bmv2, NetFPGA-SUME	[470]
P4-IPsec [471]	2020	bmv2, NetFPGA-SUME, Tofino	[472]
SPINE [473]	2019	bmv2	[474]
Qin et al. [475]	2020	bmv2	
P4NIS [476]	2020	bmv2	[477]
LANIM [478]	2020	bmv2	

P4-based firewall comprises a single MAT that allows ALLOW/DROP decision for Layer 3/4 header fields as match keys. The flow statistics are recorded with the help of counters. Another MAT allows enabling/disabling the firewall at runtime.

Vörös and Kiss [432] present a firewall implemented in P4. The parser supports Ethernet, IPv4/IPv6, UDP, and TCP headers. A ban list comprises MAC address/IP address entries that represent network hosts. Packets matching this ban list are directly dropped. To mitigate port scan or DDoS attacks, counters track packet rate and byte transfer statistics. Another MAT implements whitelist filtering.

13.2. Port Knocking

Port knocking is a simple authentication mechanism for opening network ports. Network hosts send TCP SYN packets in predefined sequences to certain ports. If the sequence is completed correctly, the server opens up a desired port. Typically, port knocking is implemented in software on servers.

P4Knocking [433] implements port knocking on P4 switches. The authors propose four different implementations for P4. In the first implementation, P4 switches track the state of knock sequences in registers where the source IP address is used as an index. The second implementation uses a CRC-hash of the source IP address as index for the knocking state registers. To resolve the problem of hash collisions, the third implementation relies on identifiers that are calculated and managed by the controller. The fourth implementation solely relies on the controller, i.e., P4 switches forward all knocking packets to the controller.

Almaini et al. [434] implement port knocking with a ticket mechanism on P4 switches. Traffic is only forwarded if the sender has a valid ticket. Predefined trusted nodes have a ticket by default, untrustworthy nodes must obtain a ticket by successful authentication via port knocking. The authors use the HIT/MISS construct of P4 as well as stateful P4 components to implement the concept. Port knocking sequences and trusted/untrusted hosts can be maintained by the control plane.

13.3. DDoS Attack Mitigation

LAMP [435] presents a cooperative mitigation mechanism for DDoS attacks that relies on information from the application layer. Ingress P4 switches add a unique identifier to the IP options header field of any processed packet. The last P4 switch ahead of the target host stores this mapping and empties the IP options header field. If a network hosts, e.g., a database server, detects an ongoing DDoS attack on the application layer, it adds an attack flag to the IP options header field and sends it back to the switch. The switch forwards this packet to the ingress switch to enable dropping of all further packets of this flow.

TDoSD@DP [436] is a P4-based mitigation mechanism for DDoS attacks targeting SIP proxies. Stateful P4 registers record the number of SIP INVITE and SIP BYE messages. Then, a simple state machine monitors sequences of INVITE and BYE messages. Many INVITES followed by zero BYE messages lead to dropping SIP INVITE packets where valid sequences of INVITE and BYE messages will keep the port open. In a follow-up work [437], the authors present an alternative approach where P4 switches act as distributed sensors. An SDN controller periodically collects data from counters of P4 switches to perform centralized attack detection. Then, attack mitigation is performed by installing DROP rules on the P4 switches.

Kuka et al. [438] present a DDoS mitigation system that targets volumetric DDoS attacks called reflective amplification attacks. The authors port an existing VHDL implementation into a P4 program that runs on FPGA targets. The implementation selects the affected subset of the incoming traffic, extracts packet data, and forwards it as a digest to an SDN controller. The SDN controller continuously evaluates this information; a heuristic algorithm identifies aggressive IP addresses by looking at the volumetric contribution of source IP addresses to the attack. In case of a detected attack, the SDN controller installs DROP rules.

Paolucci et al. [439, 440] present a stateful mitigation mechanism for TCP SYN flood attacks. It is part of a P4-based edge packet-over-optical node that also comprises traffic engineering functionality. P4 registers keep per-session statistics to detect TCP SYN flood attacks. One register records the port number of the last TCP SYN packet, the another one records the number of attempts matching the TCP SYN flood behavior. If the latter one exceeds a defined threshold, the packets are dropped.

ML-Pushback [441] proposes an extension of the Pushback DDoS attack mitigation mechanism by machine learning techniques. P4 switches implement a data collector mechanism that collects dropped packets and forwards them as digest messages to the control plane. On the control plane, a deep learning module extracts signatures and classifies the collected digest with a decision tree model. Attack mitigation is performed by throttling attacker traffic via rate limits.

Afek et al. [442] implement known mitigation mechanisms for SYN and DNS spoofing in DDoS attacks for OpenFlow and P4 targets. The OpenFlow implementation targets Open vSwitch and OpenFlow 1.5 where P4 implementations are compiled for p4c-behavioral without control plane involvement. In addition, the authors implemented a set of algorithms and methods for dynamically distributing the rule space over multiple switches.

Cardoso Lapolli et al. [443] describe an algorithmic approach to detect and stop DDoS attacks on P4 data planes. The algorithm was specifically created under the functional constraints of P4 and is based on the calculation of the Shannon entropy.

Cai et al. [445] propose a novel method for collecting traffic information to detect TCP port scanning attacks. The authors propose the "0-replacement" method as an efficient alternative to existing sampling and aggregation methods. It introduces a pending request counter (PRcounter) and relies on registers to bind hashing identifiers of the attackers' IP addresses to PRcounter values. The authors describe the concept as compliant to PSA, but only simulation results are given.

Lin et al. [446] present a comparison of OF- and P4-based implementations of basic mitigation mechanisms against SYN flooding and ARP spoofing attacks.

Musumeci et al. [447] present P4-assisted DDoS attack mitigation using an ML classifier. An ML-based DDoS attack detection module with a classifier is running on a controller. The P4 switch forwards traffic to the module; the DDoS attack detection module responds with a decision. The authors consider three use cases: packet mirroring + header mirroring + metadata extraction. In metadata extraction, P4 switches implement counters that store occurrences of IP, UDP, TCP, and SYN packets. In the case that one of the counters exceeds a defined threshold, the P4 switch inserts a custom header with the counter values and sends it to the DDoS attack detection module.

DIDA [448] presents a distributed mitigation mechanism against amplified reflection DDoS attacks. In this type of DDoS attack, spoofed requests lead to responses that are by magnitude larger. An example is a DNS ANY query. The authors rely on count-min sketch data structures and monitoring intervals to put the number of requests and responses into relation. In case of a detected DDoS attack, ACLs are used to block the traffic near to the attacker.

Dimolianis et al. [449] introduce a multi-feature DDoS detection scheme for TCP/UDP traffic. It considers the total number of incoming traffic for a particular network, the significance of the network, and the symmetry ratio of incoming and outgoing traffic for classifications. The feature analysis is time-dependent and focuses on distinct time intervals.

Scholz et al. [450] propose a SYN proxy that relies on SYN cookies or SYN authentication as protection against SYN flooding DDoS attacks. The authors present a software implementation based on DPDK and compare it to a bmv2-based P4 implementation that is ported to the T_4P_4S P4 software target, Netronome P4 hardware target, and NetFPGA SUME P4 hardware target. Evaluation results, benefits, and challenges for each platform are discussed.

Friday et al. [452] present a two-part DDoS detection and mitigation scheme. In the first part, a P4 target applies a one-way traffic analysis using bloom filters and time-dependent statistics such as moving averages. In the second part, the P4 target analyzes the bandwidth and transport protocols used by various applications to perform a volumetric analysis. The processing pipeline then decides about malicious traffic to be dropped. Administrators may supply custom network parameters used for dynamic threshold calculation that are then installed via an API on the data plane. The authors demonstrate the effectiveness of the proposed approach by three use cases: UDP amplification DDoS attacks, SYN flooding DDoS attacks, and slow DDoS attacks.

NetHide [453] prevents link-flooding attacks by obfuscating the topology of a network. It achieves this by modifying path tracing probes in the data plane while preserving their usability.

13.4. Intrusion Detection Systems (IDS) & Deep Packet Inspection (DPI)

P4ID [454] reduces intrusion detection system (IDS) processing load by apply pre-filtering on P4 switches (IDS offloading/bypassing). P4ID features a rule parser that translates Snort rules with a multistage mechanism into MAT entries. The P4 processing pipeline implements a stateless and a stateful stage. In the stateless stage, TCP/ICMP/UDP packets are matched against a MAT to decide if traffic should be dropped, forwarded to the next hop, or forwarded to the IDS. In the stateful stage, the first n packets of new flows are forwarded to the IDS. This allows that traffic targeting well-known ports can be also analyzed. Combining the feedback of the IDS for packet samples with the stateless stage is future work.

Kabasele and Sadre [455] present a two-level IDS for industrial control system (ICS) networks. The IDS targets the Modbus protocol that runs on top of TCP in SCADA networks. The first level comprises two whitelists: a flow whitelist for filtering on the TCP layer and a Modbus whitelist. If no matching entry is found for a given packet, it is forwarded to the second layer. This is in stark contrast to legacy whitelisting where packets are just dropped. In the second level, a Zeek network security analyzer acts as deep packet inspector running on a dedicated host. It analyzes the given packet, makes a decision, and instructs the controller to update filters on the switch.

DeepMatch [456] introduces deep packet inspection (DPI) for packet payloads. The concept is implemented with the help of network processors; its prototype is built with the Netronome NFP-6000 SmartNIC P4 target. The authors present regex matching capabilities that are executed in 40 Gbit/s (line rate of the platform) for stateless intra-packet matching and about 20 Gbit/s for stateful inter-packet matching. The DeepMatch functionalities are natively implemented in Micro-C for the Netronome platform and integrated into the P4 processing pipeline with the help of P4 externs.

Qin et al. [458] present an IDS based on binarized neural networks (BNN) and federated learning. BNNs compress neural networks into a simplified form that can be implemented on P4 data planes. Weights are compressed into single bits and computations, e.g., activation functions, are converted into bit-wise operations. P4 targets at the network edge then apply BNNs to classify incoming packets. To continuously train the BNNs on the P4 targets, the authors propose a federated learning scheme. Each P4 target is connected to a controller that trains an equally structured neural network with samples received from the P4 target. A cloud service aggregates local updates received from the controllers and responds with weight updates that are processed into the local model.

In the Switch-Powered Intrusion Detection (SPID) framework [460], switches compute and store flow statistics, and perform traffic change detection. If a relevant change in traffic is detected, measurement data is pushed to the control plane. In the control plane, the measurement data is fed to a ML-based anomaly detection pipeline to detect potential attacks.

13.5. Connection Security

P4-MACsec [469] presents an implementation of IEEE 802.1AE (MACsec) for P4 switches. A two-tier control plane with local switch controllers and a central controller monitor the network topology and automatically set up MACsec on detected links between P4 switches. For link discovery and monitoring, the authors implement a secured variant of LLDP that relies on encrypted payloads and sequence numbers. MACsec is directly implemented on the P4 data plane; encryption/decryption using AES-GCM is implemented on the P4 target and integrated in the P4 processing pipeline as P4 externs.

P4-IPsec [471] presents an implementation of IPsec for P4 switches. IPsec functionality is implemented in P4 and includes ESP in tunnel mode with support for different cipher suites. As in P4-MACsec, the cipher suites are implemented on the P4 target and integrated as P4 externs. In contrast to standard IPsec operation, IPsec tunnels are set up and renewed by an SDN controller without IKE. Site-to-site operation mode supports IPsec tunnels between P4 switches. Host-to-site operation mode supports roadwarrior access to an internal network via a P4 switch. To make the roadwarrior host manageable by the controller, the authors introduce a client agent tool for Linux hosts.

SPINE [473] introduces surveillance protection in the network elements by IP address obfuscation against surveillance in intermediate networks. In contrast to software-based approaches such as TOR, SPINE runs entirely on the data plane of two nodes with intermediate networks in between. It applies a one-time-pad-based encryption scheme with key rotation to encrypt IP addresses and, if present, TCP sequence and acknowledgment numbers. The SPINE nodes add a version number representing the encryption key index to each packet by which the receiving switch can select the appropriate key for decryption. The key sets required for the key rotation are maintained by a central controller.

Qin et al. [475] introduce encryption of TCP sequence numbers using substitution-boxes to protect traffic between two P4 switches. An ONOS-based controller receives the first packet of each new flow and applies security policies to decide whether the protection should be enabled. Then, it installs the necessary data in registers and updates MATs to enable TCP sequence number substitution.

P4NIS [476] proposes a scheme to protect against eavesdropping attacks. It comprises three lines of defense. In the first line of defense, packets that belong to one traffic flow are disorderly transmitted via various links. In the second line of defense, source/destination ports and sequence/acknowledgment numbers are substituted via s-boxes similar to the approach of Qin et al. [475]. The third line of defense resembles existing encryption mechanisms that are not covered by P4NIS.

LANIM [478] presents a learning-based adaptive network immune mechanism to prevent against eavesdropping attacks. It targets the Smart Identifier Network (SINET) [480], a novel, three-layer Internet architecture. LANIM applies the minimum risk ML algorithm to respond to irregular conditions and applies a policy-based encryption strategy focusing on the intent and application.

13.6. Other Fields of Application

Chang et al. [461] present IP source address encryption. It accomplishes non-linkability of IP addresses as proactive defense mechanism. Network hosts are connected to trusted P4 switches at the network edges. In between, packets are exchanged via untrusted switches/routers. The P4 switch next to the sender encrypts the sender IP address by applying an XOR operation with a hash calculated by a random number and a shared key. The P4 switch next to the receiver decrypts the original sender IP address. The mechanism includes a dynamic key update mechanism so that transformations are random.

Clé [462] proposes to upgrade particular switches in a legacy network to P4 switches that implement security network functions (SNFs) such as rule-based firewalls or IDS on P4 switches. Clé comprises a smart device upgrade selection algorithm that selects switches to be upgraded and a controller that forwards traffic streams to the P4 switches that implement SNFs.

P4DAD [463] presents a novel approach to secure duplicate address detection (DAD) against spoofing attacks. Duplicate address detection is part of NDP in IPv6 where nodes check if an IPv6 address to be applied conflicts with another node. As the messages exchanged in duplicate address detection are not authenticated or encrypted, it is vulnerable to message spoofing. As simple alternative to authentication or encryption, P4DAD introduces a mechanism to filter spoofed NDP messages. The P4 switch maintains registers to create bindings between IPv6 addresses, port numbers, and address states. Thereby, it can detect and drop spoofed NDP messages.

Chen [464] shows how AES can be implemented on Tofino-based P4 targets in P4 using MATs as lookup tables. Expansion of the AES key is performed in the control plane. MAT entries specific to the encryption keys are generated by a controller.

Gondaliya et al. [466] implement six known mechanisms against IP address spoofing for the NetFPGA SUME P4 target. Those are Network Ingress Filtering, Reverse Path Forwarding (Loose, Strict and Feasible), Spoofing Prevention Method (SPM), and Source Address Validation Improvement (SAVI). The authors compare the different mechanisms with regard to resource usage on the FPGA and report that the implementations of all mechanisms achieve a throughput of about 8.5 Gbit/s and a processing latency of about 2 µs per packet.

Poise [467] introduces context-aware policies for securing P4-based networks in BYOD scenarios. Instead of relying on a remote controller or software-based solution, Poise implements context-aware policy enforcement directly on P4 targets. Network administrators define context-aware security policies in a declarative language based on Pyretic NetCore that are then compiled into P4 programs to be executed on P4 targets. BYOD clients run a context collection module that adds context information headers to network packets. The P4 program generated by Poise then parses and uses this information to enforce ACLs based on device runtime contexts. P4 targets in Poise are managed by a Poise controller that compiles the P4 programs, installs them on the P4 targets, and provides configuration data to the collection modules. The authors present a prototype including PoiseDroid, an implementation of the context collection module for Android devices.

13.7. Summary and Analysis

Several prototypes apply P4's *custom packet headers*, e.g., for building a GTP firewall for 5G networks, a DDoS attack mitigation mechanism for the SIP, or an IDS for the Modbus protocol in industrial networks. It is also used for in-band signaling, e.g., in cooperative DDoS attack detection. All prototypes rely on *flexible packet header processing*; outstanding for this section, many of them also rely on *target-specific packet header processing functions* offered by the P4 target. Some works require custom externs, e.g., for applying MACsec or IPsec on P4 data planes. As for prototypes from the research area *Monitoring* (Section 9), many prototypes rely on registers and counters for recording statistics, e.g., for detecting attacks in DDoS mitigation or in IDSs. While custom packet headers and basic packet header processing are supported by all P4 hardware targets, the portability of prototypes using these specific functions is very limited. Several prototypes also rely on *packet processing on the control plane* where information (e.g., from blocking lists, IDS rules) is translated into MAT rules for data plane control or data received from the data plane (e.g., statistical data or packet digests) is used for runtime control. *Flexible deployment* allows to re-deploy network security programs on P4 switches in large networks.

14. Miscellaneous Applied Research Domains

This section summarizes work that falls outside of the other application domains. We describe applied research on network coding, distributed algorithms, state migration, and application support. Table 10 shows an overview of all the work described. At the end of the section, we summarize the work and analyze it with regard to P4's core features described in Section 8.1.

14.1. Network Coding

In Network Coding (NC) [517], linear encoding and decoding operations are applied on packets to increase throughput, efficiency, scalability, and resilience. Network nodes apply primitive operations, e.g., splitting, encoding, or decoding packets, to implement NC mechanisms such as multicast, forward error correction, or rerouting (resilience).

Kumar et al. [481] implement primitive NC operations such as splitting, encoding, and decoding for a PSA software switch. This is the first introduction of NC for SDN, as fixed-function data plane switches, e.g., as in OF, did not support such operations. The authors describe details of their implementation. The open-source implementation [482] relies on clone and recirculate operations to generate additional packets for encoding and decoding operations and packet processing loops. Temporary packet buffers for gathering operations are implemented with P4 registers. However, P4 hardware targets are not considered.

Gonçalves et al. [483] implement NC operations that may use information from multiple packets during processing. The authors implement their concept for PISA in $P4_{16}$. It features multiple complex NC operations that focus on multiplications in Galois fields used for encoding and decoding operations. NC operations are implemented in P4 externs that extend the capabilities of the software switch to store a specific amount of received packets. Again, hardware targets are not considered.

14.2. Distributed Algorithms

We describe related work on event processing and in-network consensus.

14.2.1. Event Processing

Data with stream characteristics often require specific processing. For example, sensor data may be analyzed to determine whether values are within certain thresholds, or chunks of data are aggregated and preprocessed.

P4CEP [484] shifts complex event processing from servers to P4 switches so that event stream data, e.g., from sensors, is directly processed on the data plane. The solution requires several workarounds to solve P4 limitations regarding stateful packet processing.

DAIET [485] introduces in-network data aggregation where the aggregation task is offloaded to the entire network. This reduces the amount of traffic and reliefs the destination of computational load. The authors provide a prototype implementation in $P4_{14}$ but only a few details are disclosed.

Sankaran et al. [486] increase the processing speed of packets by reducing the time that is required by forwarding nodes to parse the packet header. To that end, ingress routers parse the header stack to compute a so-called unique parser code (UPC) which they add to the packet header. Downstream nodes need to parse only the UPC to make forwarding decisions.

Table 10: Overview of applied research on miscellaneous research domains (Section 14).

Research work	**Year**	**Targets**	**Code**
Network Coding (Section 14.1)			
Kumar et al. [481]	2018	bmv2	[482]
Gonçalves et al. [483]	2019	bmv2	
Distributed Algorithm (Section 14.2)			
P4CEP [484]	2018	bmv2, Netronome	
DAIET [485]	2017	-	
Sankaran et al. [486]	2020	-	
Zang et al. [487]	2017	bmv2	
Dang et al. [488, 489]	2016/20	Tofino	[490]
P4BFT [491, 492]	2019	bmv2, Netronome	
SwiShmem [493]	2020	-	
SC-BFT [494]	2020	bmv2	[495]
LODGE [496]	2018	bmv2	
LOADER [497]	2020		[498]
FLAIR [499]	2020	Tofino	
State Migration (Section 14.3)			
Swing State [500]	2017	bmv2	
P4Sync [501]	2020	bmv2	[502]
Xue et al. [503]	2020	bmv2	
Kurzniar et al. [504]	2020	bmv2	
Sankaran et al. [505]	2020	NetFPGA-SUME	
Application Support (Section 14.4)			
P4DNS [506]	2019	NetFPGA SUME	[507]
P4-BNG [508]	2019	bmv2, Tofino, Netronome, NetFPGA-SUME	[509]
ARP-P4 [510]	2018	bmv2	
Glebke et al. [511]	2019	Netronome	
COIN [512]	2019	-	
Lu et al. [513]	2019	Tofino	
Yazdinejad et al. [514]	2019	bmv2	
P4rt-OVS [515]	2020	-	[516]

14.2.2. In-Network Consensus

Distributed algorithms or mechanisms may require consensus to determine the right solution or processing. This includes communication between participating entities and some ways to determine the right solution.

Zhang et al. [487] propose to offload parts of the Raft consensus algorithm to P4 switches. However, the mechanisms require an additional client to run on the switch. The authors implement their application for a P4 software switch, but details are not presented.

Dang et al. [488, 489] describe a P4 implementation of Paxos, a protocol that solves consensus for distributed algorithms in a network of unreliable processors based on information exchange between switches. This work contains a detailed description of a complex P4 implementation. The authors explain all components, provide code snippets, and discuss their design choices.

P4BFT [491, 492] introduces a consensus mechanism against buggy or malicious control plane instances. The controller responses are sent to trustworthy instances which compare the responses and establish consensus, e.g., by choosing the most common response. The authors propose to offload the comparison process to the data plane.

SwiShmem [493] is a distributed shared state management layer for the P4 data plane to implement stateful distributed network functions. In high-performance environments controllers are easily overloaded when consistency of write-intensive distributed network functions, like DDoS detection, or rate limiters, is required. Therefore, SwiShmem offloads consistency mechanisms from the control plane to the data plane. Then, consistency mechanisms operate at line rate because switches process traffic, and generate and forward state update messages without controller interaction.

Byzantine fault refers to a system where consensus between multiple entities has to be established where one or more entities are unreliable. Byzantine fault tolerance (BFT) describes mechanisms that handle such faults. However, BFTs often require significant time to reach consensus due to high computational overhead to reduce uncertainty. Switch-centric BFT (SC-BFT) [494] proposes to offload BFT functionalities, i.e., time synchronization and state synchronization, into the data plane. This significantly accelerates the consensus procedure since nodes process information at line rate.

LODGE [496] implements a mechanism for switches to make forwarding decisions based on global state without control of a central instance. Developers define global state variables which are stored by all stateful data plane devices. When such a node processes a packet that changes a global state variable, the switch generates and forwards an update packet to all other stateful switches on a predefined distribution tree. LOADER [497] introduces global state to the data plane. Consensus is maintained by the data plane devices through distributed algorithms, i.e., the switches send notification messages when global state changes. This increases scalability in comparison to mechanisms where consensus is managed by a central control entity.

FLAIR [499] accelerates read operations in leader-based consensus protocols by processing the read requests in the data plane. To that end, FLAIR devices in the core maintain persistent information about pending write operations on all objects in the system. When a client submits a read request, the FLAIR switch checks whether the requested object is stable, i.e., if it has pending write operations. If the object is stable, the FLAIR switch instructs another client with a stable version of the object, to send it to the requesting client. If the object is not stable, the FLAIR switch forwards the write request to the leader.

14.3. State Migration

In Swing State [500], switches maintain state in registers that should be migrated to other nodes. For migration, state information is carried by regular packets created by the P4 clone operation throughout the network.

P4Sync [501] is a protocol to migrate data plane state between switches. Thereby, it does not require controller interaction and provides guarantees on the authenticity of the transferred state. To that end, it leverages the switch's packet generator to transfer the content of registers between devices. Authenticity in a migration operation is guaranteed by a hash chain where each packet contains the hashed values of both the current payload and the payload of the previous packet.

Xue et al. [503] propose a hybrid approach for storing flow entries to address the issue of limited on-switch memory. While some flow entries are still stored in the internal memory of the switch, some flow entries may be stored on servers. Switches access them with only low latency via remote direct memory access (RDMA).

Kuzniar et al. [504] propose to leverage programmable switches to act as in-network cache to speed up queries over encrypted data stores. Encrypted key-value pairs are thereby stored in registers.

Sankaran et al. [505] describe a system to relieve switches from parsing headers. They propose to parse headers at an ingress switch only and add a *unique parser code* to the packet that identifies the set of headers of the packet. With this information, following switches can parse relevant information from the headers without having to parse the whole header stack.

14.4. Application Support

This subsection describes work that focuses on support or implementation of existing applications and protocols.

P4DNS [506] is an in-network DNS system. The authors propose a hybrid architecture with performance-critical components in the data plane and components with flexibility requirements in the control plane. The data plane responds to DNS requests and forwards regular traffic while cache management, recursive DNS requests, and uncached DNS responses are handled by the control plane.

P4-BNG [508] implements a carrier-grade broadband network gateway (BNG) in P4. The authors aim to provide an implementation for many different targets. To that end, they introduce a layer between data plane and control plane. This hardware-specific BNG data plane controller runs directly on the targets to provide a uniform interface to the control plane. It then configures the data plane according to the control commands from the control plane.

ARP-P4 [510] implements MAC address learning based on ARP solely on the P4 data plane. To substitute a control plane, the authors integrate MAC learning as an external function.

Glebke et al. [511] propose to offload computer vision functionalities, in particular, time-critical computations, to the data plane. To that end, the authors leverage convolution filters on a P4-programmable NIC. The necessary computations are distributed to various MATs.

COordinate-based INdexing (COIN) [512] is a mechanism to ensure efficient access to data on multiple distributed edge servers. To that end, the authors introduce a centralized instance that indexes data and its associated location. When an edge server requires data that it has not cached itself, it requests the data index at the centralized instance which provides a data location.

Lu et al. [513] propose intra-network inference (INI) and implement it in P4. It offloads neural network computations into the data plane. To that end, each P4 switch communicates via USB with a dedicated neural compute stick which performs computations.

Yazdinejad et al. [514] present a P4-based blockchain enabled packet parser. The proposed architecture focuses on FPGAs and aims to bring the security characteristics of blockchains into the data plane to greatly increase processing speed.

P4rt-OVS [515] is an extension for the OVS based on BPFs to combine the programmability of P4 and the well-known features of the OVS. P4rt-OVS enables runtime programming of the OVS, in particular, the deployment of new network features without recompilation of the OVS. It contains a P4-to-BPF compiler which allows developers to write data plane code for the OVS in P4.

14.5. Summary and Analysis

P4 facilitates the development of prototypes in the domain of network coding (see Subection 14.1) by providing *target-specific packet header processing functions*. The prototypes heavily rely on externs to implement complex packet processing behavior, i.e., encoding and decoding operations, packet splitting and packet merging. Such prototypes were mainly developed for the bmv2 and portability to hardware platforms depends on the properties of the used externs and the capabilities of the hardware targets. Distributed algorithms (see Section 14.2) leverage all sorts of P4's core features. Some prototypes *define and use custom packet headers* to transport information that are not available in standard protocols. Others rely on *flexible packet header processing* and *target-specific packet header processing functions* to implement unconventional and complex packet processing behavior. Some prototypes require *packet processing on the control plane* to resolve consistency issues or make network-wide configuration decisions. In the context of state migration (see Section 14.3) the prototypes mainly leverage externs to enable stateful processing. As a result, most projects were developed for the bmv2 with only limited portability to hardware platforms. Finally, some prototypes reimplement traditional network protocols or network elements, e.g., DNS, BNG, or ARP. Those projects mainly *define and use custom packet headers* for information transport, *flexible packet header processing* to implement the functionality of the specific protocol or network element, *target-specific packet header processing functions* for complex packet processing, and *packet processing on the control plane* for corner cases.

15. Discussion & Outlook

We discuss the findings of this survey and present an outlook.

15.1. P4 as a Language for Programmable Data Planes

From a variety of data plane programming approaches, P4 became the currently most widespread standard. Learning resources (Section 3.8) and the bmv2 P4 software target (Section 5.1) consitute low entry barriers for P4 technology. This is appealing for academia and hardware support on high speed platforms make P4 relevant for industry. The large body of literature that we surveyed in this work demonstrates that P4 has the right abstractions to build prototypes for many use cases in different application domains. Moreover, P4 allows simple and flexible definition of data plane APIs (Section 6) that can be used by simple control plane programs or complex, enterprise-grade SDN controllers. Thus, P4 allows practitioners and researchers to express their data plane and control plane algorithms in a simple way and thereby unleashes a great innovation potential. As P4 is supported by multiple platforms, there is a potentially large user group. In addition, P4 is an open programming language so that the source code can be published as open source. Therefore, public P4 code can profit from a large user community, both in quantity and quality, which is a benefit for software maintenance and security.

15.2. P4 Targets Revisited

We have listed many available P4 targets in Section 5. However, our literature overview showed that mostly the bmv2 development and testing platform and P4 hardware targets based on the Tofino ASIC were applied in the reviewed papers.

The vast majority of prototypes runs on the software switch bmv2. One reason is that it is freely available for everyone. In addition, the complexity of the code is not constrained by hardware restrictions. And finally, any required extern can be customized. Therefore, there is no limit on algorithmic complexity so that bmv2 can serve as a platform for any use case – but only from a functional point of view. As it is a pure software-based prototyping solution, it cannot provide high throughput and is, therefore, not suitable for deployment in productive environments.

The Tofino ASIC is the base for P4 hardware targets with high throughput on many ports. It is currently the only available programmable data plane platform with throughput rates over 12.8 Tbit/s and ports running at up to 400 Gbit/s. Therefore, Tofino-based devices are appropriate programmable data planes for production environments like data centers or core networks. Tofino uses P4 as native programming language. Therefore, comprehensive tools are offered to support the P4 development process on this platform. Moreover, P4 gives access to all features of the Tofino chip so that there is no penalty of using P4 as a programming language. Existing restrictions are due to the functional limitation of a high speed platform. Thus, prototypes for Tofino are more challenging but prove the technical feasibility of a new concept at commercial scale. Probably for these reasons the Tofino turned out to be the mostly used hardware platform in our survey.

P4 can be also used on FPGA- or NPU-based targets. They come with only a few ports and lower throughput rates so that they may be used for special-purpose server applications but not for typical switching devices. They excel through the possibility to extend the target functionality with user-defined externs. These cards are typically programmed by vendor-specific languages. P4 support is achieved by trans-compilers that translate P4 programs into the vendor-specific format. P4 programmability might be limited to a restricted feature set while access to all features of a target is only possible through the vendor-specific programming language. Whether the application of P4 for such targets is beneficial compared to vendor-specific programming languages or interfaces mainly depends on the use case, level of knowledge of the programmer, and if prospect target-independence is a goal.

15.3. Target Independence and Portability

Many of the surveyed works profited from P4's core features that we summarized in Section 8. Often P4 programs were developed only for the bmv2 target due to the complexity of their algorithm, required interaction with fixed-function blocks, or dependence on custom extern functions. The portability of such programs is limited to platforms with similar externs and even then the code needs to be significantly adapted.

In some use cases, the authors even miss the original objectives of P4. They suggest P4 for complex *packet processing* operations while P4 has been primarily conceived for *packet header processing* with simple operations on high speed data planes.

Although some of the presented prototypes may not be portable to current P4 hardware targets, they are close to modern switch architectures as their overall pipeline is described in P4. Thereby, the conceptual feasibility of new data plane algorithms can be proven. This is an advantage of bmv2-based prototypes compared to general software implementations.

15.4. A Business Perspective for P4-Programmable Data Planes

Today, the most prevalent hardware network appliances are proprietary devices for which customized hardware and software are jointly developed.

Data plane programming breaks with this process. Programmable packet processing ASICs such as the Tofino may be sold by specialized manufacturers and integrated by other vendors with a motherboard, CPU, memory, and connectors in white box switches. The accompanying software, i.e., data plane and control plane programs, might be provided by the same vendor, a third party, or implemented by the users themselves.

Because software is developed independently of hardware, the agility of the development process can be increased, which can reduce the time to market. Hardware platforms become reusable; they can be leveraged for multiple purposes with the help of appropriate P4 programs.

Network solution providers may leverage the lowered entry barrier for customized hardware appliances to develop and sell P4 software for various P4-capable targets, at least with moderate adaptation effort. A decade of implementation experience may no longer be a prerequisite for that business.

In addition, companies with large networks and particular use cases, e.g., special applications in data centers, may use customized algorithms to overcome inefficiencies of standardized protocols or mechanisms.

Large companies can avoid vendor lock-in by acquisition of programmable components instead of black boxes. The components are assembled possibly with open-source software leveraging data plane programming, SDN, and NFV. The ACCESS 4.0 architecture [518] and the O-RAN Alliance [519] are examples. This type of disaggregation also enables cost scaling effects where off-the-shelf components are bought at moderate cost instead of expensive specialized appliances.

15.5. Outlook

P4 is primarily a programming language for high-speed switches. Currently, it is supported by Intel's Tofino ASIC, but other manufacturers already announced support for P4 for the future.

The many prototypes surveyed in this paper showed that there is a need for more functionality on programmable switches, which may be provided by extern functions. While they reduce portability, they enable more use cases. Examples for such extern functions are features that have been used in some of the pure software-based P4 prototypes. They encrypt and decrypt packet payload, support floating-point operations, provide flexible hash functions, or allow more complex calculations. Those externs might be provided by the target manufacturers for common use cases or integrated by users.

Hardware with a vendor-specific programming language may benefit from offering interfaces and cross-compilers for P4 together with useful extern functions. Although this may not give access to the full functionality of the platform, users with P4 programming knowledge can customize such devices for their needs without worrying about hardware details.

The biggest driver for P4 is possibly disaggregation. While currently devices from different vendors can be orchestrated by a customized controller, P4 may have the potential to extend disaggregation towards specialized appliances based on off-the-shelf programmable hardware. Hardware without an open programming interface cannot profit from that market.

16. Conclusion

In this paper, we first gave a tutorial on data plane programming with P4. We delineated it from SDN and introduced programming models with a special focus on PISA which is most relevant for P4. We provided an overview of the current state of P4 with regard to programming language, architectures, compilers, targets, and data plane APIs. We reported research efforts to advance P4 that fall in the areas of optimization of development and deployment, research on P4 targets, and P4-specific approaches for control plane operation.

In the second part of the paper, we analyzed 245 papers on applied research that leverage P4 for implementation purposes. We categorized these publications into research domains, summarized their key points, and characterized them by prototype, target platform, and source code availability. For each research domain, we presented an analysis on how works benefit from P4. To that end, we identified a small set of core features that facilitate implementations. The survey proved a tremendous uptake of P4 for prototyping in academic research from 2018 to 2021. One reason is certainly the multitude of openly available resources on P4 and the bmv2 P4 software target. They are an ideal starting point for creating P4-based prototypes, even for beginners.

The many P4-based activities which emerged only within short time show that P4 technology can speed up the evolution of computer networking. While multiple hardware targets are available, most hardware-based prototypes leverage the Tofino ASIC that is optimized for high throughput on many ports and particularly suited for data center and WAN applications. However, the majority of P4-based prototypes was implemented with the bmv2 software switch. Many of them were not ported to hardware, probably due to the complexity of their data plane algorithms and lack of required extern functions on current hardware. This may change in the future if new P4 hardware targets are available. We expect P4 to become a base technology for multiple hardware appliances, in particular in the context of disaggregation and for small-scale markets.

17. Acknowledgement

This work was partly supported by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/1-2. The authors alone are responsible for the content of this paper.

References

[1] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. F. Kaashoek, The Click Modular Router, ACM Transactions on Computer Systems (TOCS) 18 (2000) 217–231.

[2] VPP/What is VPP?, `https://bit.ly/2mrxVGE`, accessed 01-20-2021 (2021).

[3] GitHub: NPL-Spec, `https://github.com/nplang/NPL-Spec`, accessed 01-20-2021 (2021).

[4] Software Defined Specification Environment for Networking (SDNet), `https://www.xilinx.com/support/documentation/backgrounders/sdnet-backgrounder.pdf`, accessed 01-20-2021 (2021).

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker, P4: Programming Protocol-independent Packet Processors, ACM SIGCOMM Computer Communications Review (CCR) 44 (2014) 87–95.

[6] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, T. Turletti, A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks, IEEE Communications Surveys & Tutorials (COMST) 16 (2014) 1617–1634.

[7] Y. Jarraya, T. Madi, M. Debbabi, A Survey and a Layered Taxonomy of Software-Defined Networking, IEEE Communications Surveys & Tutorials (COMST) 16 (2014) 1955–1980.

[8] W. Xia, Y. Wen, C. H. Foh, D. Niyato, H. Xie, A Survey on Software-Defined Networking, IEEE Communications Surveys & Tutorials (COMST) 17 (2015) 27–51.

[9] D. F. Macedo, D. Guedes, L. F. M. Vieira, M. A. M. Vieira, M. Nogueira, Programmable Networks—From Software-Defined Radio to Software-Defined Networking, IEEE Communications Surveys & Tutorials (COMST) 17 (2015) 1102–1125.

[10] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, S. Uhlig, Software-Defined Networking: A Comprehensive Survey, Proceedings of the IEEE 103 (2015) 14–76.

[11] R. Masoudi, A. Ghaffari, Software defined networks: A survey, Journal of Network and Computer Applications (JNCA) 67 (2016) 1–25.

[12] C. Trois, M. D. Del Fabro, L. C. E. de Bona, M. Martinello, A Survey on SDN Programming Languages: Toward a Taxonomy, IEEE Communications Surveys & Tutorials (COMST) 18 (2016) 2687–2712.

[13] W. Braun, M. Menth, Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices, MDPI Future Internet Journal (FI) 6 (2014) 302–336.

[14] F. Hu, Q. Hao, K. Bao, A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation, IEEE Communications Surveys & Tutorials (COMST) 16 (2014) 2181–2206.

[15] A. Lara, A. Kolasani, B. Ramamurthy, Network Innovation using OpenFlow: A Survey, IEEE Communications Surveys & Tutorials (COMST) 16 (2014) 493–512.

[16] R. Bifulco, G. Rétvári, A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems, in: IEEE International Conference on High Performance Switching and Routing (HPSR), 2018, pp. 1–7.

[17] E. Kaljic, A. Maric, P. Njemcevic, M. Hadzialic, A Survey on Data Plane Flexibility and Programmability in Software-Defined Networking, IEEE ACCESS 7 (2019) 47804–47840.

[18] O. Michel, R. Bifulco, G. Rétvári, S. Schmid, The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications, ACM Computing Surveys 1 (2021).

[19] S. Kaur, K. Kumar, N. Aggarwal, A review on p4-programmable data planes: Architecture, research efforts, and future directions, Computer Communications 170 (2021).

[20] E. F. Kfoury, J. Crichigno, E. Bou-Harb, An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends, ArXiv e-prints (2021).

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: Enabling Innovation in Campus Networks, ACM SIGCOMM Computer Communications Review (CCR) 38 (2008) 69–74.

[22] BESS: Berkeley Extensible Software Switch, `http://span.cs.berkeley.edu/bess.html`, accessed 01-20-2021 (2021).

[23] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, M. Horowitz, Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN, ACM SIGCOMM Conference 43 (2013) 99–110.

[24] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, T. Edsall, DRMT: Disaggregated Programmable Switching, in: ACM SIGCOMM Conference, 2017, p. 1–14.

[25] Google Presentations: P4 Tutorial, `http://bit.ly/p4d2-2018-spring`, accessed 01-20-2021 (2018).

[26] Website of the P4 Language Consortium, `https://p4.org/`, accessed 01-20-2021 (2021).

[27] The P4 Language Specification, https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf, accessed 01-20-2021 (2018).

[28] P4 16 Language Specification (v.1.2.1, https://p4.org/p4-spec/docs/P4-16-v1.2.1.html, accessed 01-20-2021 (2020).

[29] M. Moshref, A. Bhargava, A. Gupta, M. Yu, R. Govindan, Flow-level State Transition as a New Switch Primitive for SDN, in: ACM SIGCOMM Conference, 2014, p. 61–66.

[30] G. Bianchi, M. Bonola, A. Capone, C. Cascone, OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch, ACM SIGCOMM Computer Communications Review (CCR) 44 (2014) 44–51.

[31] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, S. Licking, Packet Transactions: High-Level Programming for Line-Rate Switches, in: ACM SIGCOMM Conference, 2016, p. 15–28.

[32] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, G. Siracusano, FlowBlaze: Stateful Packet Processing in Hardware, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2019, p. 531–547.

[33] H. Song, Protocol-Oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane, in: ACM Workshop on Hot Topics in Networks (HotNets), 2013, p. 127–132.

[34] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, D. Walker, NetKAT: Semantic Foundations for Networks, in: ACM Symposium on Principles of Programming Languages (POPL), 2014, p. 113–126.

[35] P4 Tutorial, https://github.com/p4lang/tutorials, accessed 05-05-2021 (2021).

[36] P4 Guide, https://github.com/jafingerhut/p4-guide, accessed 05-05-2021 (2021).

[37] P4 Learning, https://github.com/nsg-ethz/p4-learning, accessed 05-05-2021 (2021).

[38] Charter of the P4 Architecture WG, https://github.com/p4lang/p4-spec/blob/master/p4-16/psa/charter/P4_Arch_Charter.mdk, accessed 01-20-2021 (2021).

[39] P4_16 PSA Specification (v1.1), https://p4lang.github.io/p4-spec/docs/PSA-v1.1.0.html, accessed 01-20-2021 (2018).

[40] P4-HLIR Specification v.0.9.30, https://github.com/p4lang/p4-hlir/blob/master/HLIRSpec.pdf, accessed 01-20-2021 (2016).

[41] GitHub: p4c, https://github.com/p4lang/p4c, accessed 01-20-2021 (2021).

[42] P. G. Patra, C. E. Rothenberg, G. Pongracz, MACSAD: High Performance Dataplane Applications on the Move, in: IEEE International Conference on High Performance Switching and Routing (HPSR), 2017, pp. 1–6.

[43] Open Data Plane, https://opendataplane.org/, accessed 01-20-2021 (2021).

[44] L. Jose, M. R. N. M. Lisa Yan, Stanford University; George Varghese, Compiling Packet Programs to Reconfigurable Switches, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2015, p. 103–115.

[45] P. Li, Y. Luo, P4GPU: Accelerate Packet Processing of a P4 Program with a CPU-GPU Heterogeneous Architecture, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2016, pp. 125–126.

[46] GitHub: p4c-behavioural, https://github.com/p4lang/p4c-behavioral/tree/master/p4c_bm, accessed 01-20-2021 (2021).

[47] GitHub: Behavioural Model Version 2 (BMv2), https://github.com/p4lang/behavioral-model, accessed 01-20-2021 (2021).

[48] P4 Behaviour Model: Why did we need BMv2, https://github.com/p4lang/behavioral-model#why-did-we-replace-p4c-behavioral-with-bmv2, accessed 01-20-2021 (2021).

[49] GitHub: Behavioral model targets, https://github.com/p4lang/behavioral-model/blob/master/targets/README.md, accessed 01-20-2021 (2021).

[50] BMv2 Performance, https://github.com/p4lang/behavioral-model/blob/master/docs/performance.md, accessed 01-20-2021 (2021).

[51] GitHub: eBPF Backend for p4c, https://github.com/p4lang/p4c/tree/master/backends/ebpf, accessed 01-20-2021 (2021).

[52] p4c-ubpf: a New Back-end for the P4 Compiler, https://p4.org/p4/p4c-ubpf.html, accessed 01-20-2021 (2021).

[53] GitHub: p4c-xdp, https://github.com/vmware/p4c-xdp, accessed 01-20-2021 (2021).

[54] P4@ELTE, http://p4.elte.hu/, accessed 01-20-2021 (2021).

[55] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, M. Tejfel, High speed packet forwarding compiled from protocol independent data plane specifications, in: ACM SIGCOMM Conference, 2016, p. 629–630.

[56] Data Plane Development Kit (DPDK), `https://www.dpdk.org/`, accessed 01-20-2021 (2021).

[57] GitHub: T4P4S, `https://github.com/P4ELTE/t4p4s`, accessed 01-20-2021 (2021).

[58] A. Bhardwaj, A. Shree, V. B. Reddy, S. Bansal, A Preliminary Performance Model for Optimizing Software Packet Processing Pipelines, in: ACM SIGOPS Asia-Pacific Workshop on System (APSys), 2017, pp. 1–7.

[59] X. Wu, P. Li, T. Miskell, L. Wang, Y. Luo, X. Jiang, Ripple: An Efficient Runtime Reconfigurable P4 Data Plane for Multicore Systems, in: International Conference on Networking and Network Applications (NaNA), 2019, pp. 142–148.

[60] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, J. Rexford, PISCES: A Programmable, Protocol-Independent Software Switch, in: ACM SIGCOMM Conference, 2016, p. 525–538.

[61] Open vSwitch, `https://www.openvswitch.org/`, accessed 01-20-2021 (2021).

[62] GitHub: PISCES, `https://github.com/P4-vSwitch`, accessed 01-20-2021 (2021).

[63] S. Choi, X. Long, M. Shahbaz, S. Booth, A. Keep, J. Marshall, C. Kim, The Case for a Flexible Low-Level Backend for Software Data Planes, in: Asia-Pacific Workshop on Networking (APnet), 2017, p. 71–77.

[64] S. Choi, X. Long, M. Shahbaz, S. Booth, A. Keep, J. Marshall, C. Kim, PVPP: A Programmable Vector Packet Processor, in: ACM Symposium on SDN Research (SOSR), 2017, p. 197–198.

[65] Northbound Networks - Who are You?, `https://northboundnetworks.com/pages/about-us`, accessed 01-20-2021 (2021).

[66] GitHub: ZodiacFX-P4, `https://github.com/NorthboundNetworks/ZodiacFX-P4`, accessed 01-20-2021 (2021).

[67] GitHub: p4c-zodiacfx, `https://github.com/NorthboundNetworks/p4c-zodiacfx`, accessed 01-20-2021 (2021).

[68] P. Zanna, P. Radcliffe, K. G. Chavez, A Method for Comparing OpenFlow and P4, in: International Telecommunication Networks and Applications Conference (ITNAC), 2019, pp. 1–3.

[69] GitHub: P4-NetFPGA, https://github.com/NetFPGA/P4-NetFPGA-public/wiki, accessed 01-20-2021 (2021).

[70] S. Ibanez, G. Brebner, N. McKeown, N. Zilberman, The P4-NetFPGA Workflow for Line-Rate Packet Processing, in: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2019, p. 1–9.

[71] N. Zilberman, Y. Audzevich, G. A. Covington, A. W. Moore, NetFPGA SUME: Toward 100 Gbps as Research Commodity, IEEE Micro 34 (2014) 32–41.

[72] Netcope P4, https://www.netcope.com/Netcope/media/content/NetcopeP4_2019_web.pdf, accessed 01-20-2021 (2021).

[73] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, H. Weatherspoon, P4FPGA: A Rapid Prototyping Framework for P4, in: ACM Symposium on SDN Research (SOSR), 2017, p. 122–135.

[74] GitHub: P4FPGA, https://github.com/p4fpga/p4fpga, accessed 01-20-2021 (2021).

[75] P. Benácek, V. Pu, H. Kubátová, P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers, in: IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2016, pp. 148–155.

[76] P. Benáček, V. Puš, J. Kořenek, M. Kekely, Line Rate Programmable Packet Processing in 100Gb Networks, in: International Conference on Field Programmable Logic and Applications (FPL), 2017, pp. 1–1.

[77] J. Cabal, P. Benáček, L. Kekely, M. Kekely, V. Puš, J. Kořenek, Configurable FPGA Packet Parser for Terabit Networks with Guaranteed Wire-Speed Throughput, in: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2018, p. 249–258.

[78] S. da Silva, Jeferson, Boyer, François-Raymond, Langlois, J. Pierre, P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs, in: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2018, p. 147–152.

[79] M. Kekely, J. Korenek, Mapping of P4 Match Action Tables to FPGA, in: International Conference on Field Programmable Logic and Applications (FPL), 2017, pp. 1–2.

[80] R. Iša, P. Benáček, V. Puš, Verification of Generated RTL from P4 Source Code, in: IEEE International Conference on Network Protocols (ICNP), 2018, pp. 444–445.

[81] Z. Cao, H. Su, Q. Yang, J. Shen, M. Wen, C. Zhang, P4 to FPGA-A Fast Approach for Generating Efficient Network Processors, IEEE ACCESS 8 (2020) 23440–23456.

[82] Z. Cao, H. Su, Q. Yang, M. Wen, C. Zhang, A Template-based Framework for Generating Network Processor in FPGA, in: IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2019, pp. 1057–1058.

[83] Open Tofino, https://github.com/barefootnetworks/open-tofino, accessed 01-22-2021 (2021).

[84] EdgeCore Wedge 100BF-32X, https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335, accessed 01-20-2021 (2021).

[85] APS Networks BF2556X-1T-A1F, https://stordirect.com/shop/switches/25g-switches/aps-networks-bf2556x-1t-a1f/, acessed 01-22-2021 (2021).

[86] APS Networks BF6064X-T-A2F, https://stordirect.com/shop/switches/100g-switches/aps-networks-bf6064x-t-a2f/, acessed 01-22-2021 (2021).

[87] Netberg Aurora 610, https://netbergtw.com/products/aurora-610/, accessed 01-20-2021 (2021).

[88] Arista Press Release: Arista Announces New Multi-function Platform for Cloud Networking, https://www.arista.com/en/company/news/press-release/5148-pr-20180605, accessed 01-20-2021 (2021).

[89] Cisco Blog: Increase Flexibility with Cisco's Programmable Cloud Infrastructure, https://blogs.cisco.com/datacenter/increase-flexibility-with-ciscos-programmable-cloud-infrastructure, accessed 01-20-2021 (2021).

[90] SONiC - Supported Platforms, https://azure.github.io/SONiC/Supported-Devices-and-Platforms.html, accessed 01-20-2021 (2021).

[91] A. Seibulescu, M. Baldi, Leveraging P4 Flexibility to Expose Target-Specific Features, in: P4 Workshop in Europe (EuroP4), 2020, p. 36–42.

[92] The Pensando Distributed Services Platform, https://pensando.io/our-platform/, accessed 01-20-2021 (2021).

[93] Netronome: P4 Data Plane Programming, https://netronome.com/media/documents/WP_P4_Data_Plane_Programming.pdf, accessed 01-20-2021 (2018).

[94] Netronome: Programming with P4 and C, https://www.netronome.com/media/documents/WP_Programming_with_P4_and_C.pdf, accessed 09-20-2019 (2018).

[95] H. Harkous, M. Jarschel, M. He, R. Pries, W. Kellerer, Towards Understanding the Performance of P4 Programmable Hardware, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2019, pp. 1–6.

[96] Apache Thrift, https://thrift.apache.org/, accessed 01-20-2021 (2021).

[97] gRPC, https://grpc.io/, accessed 01-20-2021 (2021).

[98] Google Protocol Buffers, https://developers.google.com/protocol-buffers/, accessed 01-20-2021 (2021).

[99] Charter of the P4 API WG, https://github.com/p4lang/p4-spec/blob/master/api/charter/P4_API_WG_charter.mdk, accessed 01-20-2021 (2021).

[100] P4 Runtime API Specification v.1.3.0 (2019-12-01), https://p4.org/p4runtime/spec/v1.3.0/P4Runtime-Spec.html, accessed 01-20-2021 (2020).

[101] ONOS: P4 brigade, https://wiki.onosproject.org/display/ONOS/P4+brigade, accessed 01-20-2021 (2021).

[102] OpenDaylight: P4 brigade, P4PluginDeveloperGuide, accessed 09-23-2019 (2019).

[103] B. O'Connor, Y. Tseng, M. Pudelko, C. Cascone, A. Endurthi, Y. Wang, A. Ghaffarkhah, D. Gopalpur, T. Everman, T. Madejski, J. Wanderer, A. Vahdat, Using P4 on Fixed-Pipeline and Programmable Stratum Switches, in: P4 Workshop in Europe (EuroP4), 2010, pp. 1–2.

[104] GitHub: P4tutorial, https://github.com/p4lang/tutorials/tree/master/utils/p4runtime_lib, accessed 01-20-2021 (2021).

[105] GitHub: PI Library, https://github.com/p4lang/PI, accessed 01-20-2021 (2021).

[106] GitHub: Behavioural Model - simple_switch_grpc, https://github.com/p4lang/behavioral-model/tree/master/targets/simple_switch_grpc, accessed 01-20-2021 (2021).

[107] GitHub: bmv2 Runtime CLI, https://github.com/p4lang/behavioral-model/blob/master/tools/runtime_CLI.py, accessed 01-20-2021 (2021).

[108] E. O. Zaballa, Z. Zhou, Graph-to-P4: A P4 Boilerplate Code Generator for Parse Graphs, in: P4 Workshop in Europe (EuroP4), 2019, pp. 1–2.

[109] Y. Zhou, J. Bi, ClickP4: Towards Modular Programming of P4, in: ACM SIGCOMM Conference Posters and Demos, 2017, p. 100–102.

[110] M. Baldi, daPIPE A Data Plane Incremental Programming Environment, in: P4 Workshop in Europe (EuroP4), 2019, pp. 1–6.

[111] M. Eichholz, E. Campbell, N. Foster, G. Salvaneschi, M. Mezini, How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4, in: European Conference on Object-Oriented Programming (ECOOP), 2019, pp. 1–28.

[112] M. Riftadi, F. Kuipers, P4I/O: Intent-Based Networking with P4, in: IEEE Conference on Network Softwarization (NetSoft), 2019, pp. 438–443.

[113] L. Yu, J. Sonchack, V. Liu, Mantis: Reactive Programmable Switches, in: ACM SIGCOMM Conference, 2020, p. 296–309.

[114] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, M. Yu, Lyra: A Cross-Platform Language and Compiler for Data PlaneProgramming on Heterogeneous ASICs, in: ACM SIGCOMM Conference, 2020, p. 435–450.

[115] M. Riftadi, J. Oostenbrink, F. Kuipers, GP4P4: Enabling Self-Programming Networks, ArXiv e-prints (2019).

[116] D. Moro, D. Sanvito, A. Capone, FlowBlaze.p4: a library for quick prototyping of stateful SDN applications in P4, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2020, pp. 95–99.

[117] D. Moro, D. Sanvito, A. Capone, Demonstrating FlowBlaze.p4: fast prototyping for EFSM-based data plane applications, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2020, pp. 116–117.

[118] D. Moro, D. Sanvito, A. Capone, Developing EFSM-Based Stateful Applications with FlowBlaze.P4 and ONOS, in: P4 Workshop in Europe (EuroP4), 2020, p. 52–53.

[119] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, B. T. Loo, Flightplan: Dataplane disaggregation and placement for p4 programs, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2021, pp. 571–592.

[120] R. Shah, A. Shirke, A. Trehan, M. Vutukuru, P. Kulkarni, pcube: Primitives for Network Data Plane Programming, in: IEEE International Conference on Network Protocols (ICNP), 2018, pp. 430–435.

[121] Z. Ma, J. Bi, C. Zhang, Y. Zhou, A. B. Dogar, CacheP4: A Behavior-level Caching Mechanism for P4, in: ACM SIGCOMM Conference Posters and Demos, 2017, p. 108–110.

[122] A. Abhashkumar, J. Lee, J. Tourrilhes, S. Banerjee, W. Wu, J.-M. Kang, A. Akella, P5: Policy-driven Optimization of P4 Pipeline, in: ACM Symposium on SDN Research (SOSR), 2017, p. 136–142.

[123] P. Wintermeyer, M. Apostolaki, A. Dietmüller, L. Vanbever, P2GO: P4 Profile-Guided Optimizations, in: ACM Workshop on Hot Topics in Networks (HotNets), 2020, p. 146–152.

[124] S. Yang, L. Baia, L. Cui, Z. Ming, Y. Wu, S. Yu, H. Shen, Y. Pan, P4 Edge node enabling stateful traffic engineering and cyber security, Journal of Network and Computer Applications (JNCA) 171 (2020) A84–A95.

[125] B. Vass, E. Bérczi-Kovács, C. Raiciu, G. Rétvári, Compiling Packet Programs to Reconfigurable Switches: Theory and Algorithms, in: P4 Workshop in Europe (EuroP4), 2020, p. 28–35.

[126] S. Abdi, U. Aftab, G. Bailey, B. Boughzala, F. Dewal, S. Parsazad, E. Tremblay, PFPSim: A Programmable Forwarding Plane Simulator, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2016, pp. 55–60.

[127] J. Bai, J. Bi, P. Kuang, C. Fan, Y. Zhou, C. Zhang, NS4: Enabling Programmable Data Plane Simulation, in: ACM Symposium on SDN Research (SOSR), 2018, pp. 1–7.

[128] C. Fan, J. Bi, Y. Zhou, C. Zhang, H. Yu, NS4: A P4-Driven Network Simulator, in: ACM SIGCOMM Conference Posters and Demos, 2017, p. 105–107.

[129] N. McKeown, D. Talayco, G. Varghese, N. P. Lopes, N. Bjørner, A. Rybalchenko, Automatically Verifying Reachability and Well-Formedness in P4 Networks, `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/09/p4nod.pdf`, accessed 01-20-2021 (2016).

[130] A. Kheradmand, G. Rosu, P4K: A Formal Semantics of P4 and Applications, ArXiv e-prints (2018).

[131] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, N. Foster, P4V: Practical Verification for Programmable Data Planes, in: ACM SIGCOMM Conference, 2018, p. 490–503.

[132] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, M. Barcellos, Uncovering Bugs in P4 Programs with Assertion-based Verification, in: ACM Symposium on SDN Research (SOSR), 2018, p. 1–7.

[133] M. Neves, L. Freire, A. Schaeffer-Filho, M. Barcellos, Verification of P4 Programs in Feasible Time using Assertions, in: ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2018, p. 73–85.

[134] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, C. Raiciu, Debugging P4 Programs with Vera, in: ACM SIGCOMM Conference, 2018, p. 518–532.

[135] M. A. Noureddine, A. Hsu, M. Caesar, F. A. Zaraket, W. H. Sanders, P4AIG: Circuit-Level Verification of P4 Programs, in: IEEE/IFIP International Conference on Dependable Systems and Networks – Supplemental Volume (DSN-S), 2019, pp. 21–22.

[136] D. Dumitrescu, R. Stoenescu, L. Negreanu, C. Raiciu, bf4: towards bug-free P4 programs, in: ACM SIGCOMM Conference, 2020, p. 571–585.

[137] D. Dumitrescu, R. Stoenescu, M. Popovici, L. Negreanu, C. Raiciu, Dataplane equivalence and its applications, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2019, pp. 683–698.

[138] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, A. Akella, Liveness Verification of Stateful Network Functions, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2020, pp. 257–272.

[139] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, P. Athanas, P4Pktgen: Automated Test Case Generation for P4 Programs, in: ACM Symposium on SDN Research (SOSR), 2018, pp. 1–7.

[140] Y. Zhou, J. Bi, Y. Lin, Y. Wang, D. Zhang, Z. Xi, J. Cao, C. Sun, P4Tester: Efficient Runtime Rule Fault Detection for Programmable Data Planes, in: IEEE International Workshop on Quality of Service (IWQoS), 2019, pp. 1–10.

[141] GitHub: P4app, `https://github.com/p4lang/p4app`, accessed 01-20-2021 (2021).

[142] A. Shukla, K. N. Hudemann, A. Hecker, S. Schmid, Runtime Verification of P4 Switches with Reinforcement Learning, in: Workshop on Network Meets AI & ML, 2019, p. 1–7.

[143] D. Jindal, R. Joshi, B. Leong, P4TrafficTool: Automated Code Generation for P4 Traffic Generators and Analyzers, in: ACM Symposium on SDN Research (SOSR), 2019, p. 152–153.

[144] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, H. Weatherspoon, Whippersnapper: A P4 Language Benchmark Suite, in: ACM Symposium on SDN Research (SOSR), 2017, p. 95–101.

[145] F. Rodriguez, P. G. K. Patra, L. Csikor, C. E. Rothenberg, P. Vörös, S. Laki, G. Pongrácz, BB-Gen: A Packet Crafter for P4 Target Evaluation, in: ACM SIGCOMM Conference Posters and Demos, 2018, p. 111–113.

[146] H. Harkous, M. Jarschel, M. He, R. Pries, W. Kellerer, P8: P4 with Predictable Packet Processing Performance, IEEE Transactions on Network and Service Management (TNSM) (2020) 1–1.

[147] S. Kodeswaran, M. T. Arashloo, P. Tammana, J. Rexford, Tracking P4 Program Execution in the Data Plane, in: ACM Symposium on SDN Research (SOSR), 2020, p. 117–122.

[148] K. Birnfeld, D. C. da Silva, W. Cordeiro, B. B. N. de França, P4 Switch Code Data Flow Analysis: Towards Stronger Verification of Forwarding Plane Software, in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2020, pp. 1–8.

[149] M. Neves, B. Huffaker, K. Levchenko, M. Barcellos, Dynamic Property Enforcement in Programmable Data Planes, in: IFIP-TC6 Networking Conference (Networking), 2019, pp. 1–9.

[150] C. Zhang, J. Bi, Y. Zhou, J. Wu, B. Liu, Z. Li, A. B. Dogar, Y. Wang, P4DB: On-the-fly Debugging of the Programmable Data Plane, in: IEEE International Conference on Network Protocols (ICNP), 2017, pp. 1–10.

[151] Y. Zhou, J. Bi, C. Zhang, B. Liu, Z. Li, Y. Wang, M. Yu, P4DB: On-the-Fly Debugging for Programmable Data Planes, IEEE/ACM Transactions on Networking (ToN) 27 (2019) 1714–1727.

[152] M. Neves, K. Levchenko, M. Barcellos, Sandboxing Data Plane Programs for Fun and Profit, in: ACM SIGCOMM Conference Posters and Demos, 2017, p. 103–104.

[153] A. Shukla, S. Fathalli, T. Zinner, A. Hecker, S. Schmid, P4Consist: Toward Consistent P4 SDNs, IEEE Journal on Selected Areas in Communications (JSAC) 38 (2020) 1293–1307.

[154] Z. Xia, J. Bi, Y. Zhou, C. Zhang, KeySight: A Scalable Troubleshooting Platform Based on Network Telemetry, in: ACM Symposium on SDN Research (SOSR), 2018, pp. 1–2.

[155] F. Ruffy, T. Wang, A. Sivaraman, Gauntlet: Finding Bugs in Compilers for Programmable Packet Processing, in: USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2020, pp. 1–17.

[156] J. Krude, J. Hofmann, M. Eichholz, K. Wehrle, A. Koch, M. Mezini, Online Reprogrammable Multi Tenant Switches, in: ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms, 2019, p. 1–8.

[157] D. Hancock, J. van der Merwe, HyPer4: Using P4 to Virtualize the Programmable Data Plane, in: ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2016, p. 35–49.

[158] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, J. Wu, HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane, in: IEEE International Conference on Computer Communications and Networks (ICCCN), 2017, pp. 1–9.

[159] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, J. Wu, MPVisor: A Modular Programmable Data Plane Hypervisor, in: ACM Symposium on SDN Research (SOSR), 2017, p. 179–180.

[160] GitHub: HyperVDP, `https://github.com/HyperVDP`, accessed 01-20-2021 (2021).

[161] C. Zhang, J. Bi, Y. Zhou, J. Wu, HyperVDP: High-Performance Virtualization of the Programmable Data Plane, IEEE Journal on Selected Areas in Communications (JSAC) 37 (2019) 556–569.

[162] M. Saquetti, G. Bueno, W. Cordeiro, J. R. Azambuja, P4VBox: Enabling P4-Based Switch Virtualization, IEEE Communications Letters 24 (2020) 146–149.

[163] M. Saquetti, G. Bueno, W. Cordeiro, J. R. Azambuja, VirtP4: An Architecture for P4 Virtualization, in: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019, pp. 75–78.

[164] P. Zheng, T. Benson, C. Hu, P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs, in: ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2018, p. 98–111.

[165] R. Parizotto, L. Castanheira, F. Bonetti, A. Santos, A. Schaeffer-Filho, PRIME: Programming In-Network Modular Extensions, in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2020, pp. 1–9.

[166] E. O. Zaballa, D. Franco, M. S. Berger, M. Higuero, A Perspective on P4-Based Data and Control Plane Modularity for Network Automation, in: P4 Workshop in Europe (EuroP4), 2020, p. 59–61.

[167] R. Stoyanov, N. Zilberman, MTPSA: Multi-Tenant Programmable Switches, in: P4 Workshop in Europe (EuroP4), 2020, p. 43–48.

[168] GitHub: MTPSA, `https://github.com/mtpsa`, accessed 01-20-2021 (2021).

[169] S. Han, S. Jang, H. Choi, H. Lee, S. Pack, Virtualization in Programmable Data Plane: A Survey and Open Challenges, IEEE Open Journal of the Communications Society 1 (2020) 527–534.

[170] J. Santiago da Silva, T. Stimpfling, T. Luinaud, B. Fradj, B. Boughzala, One for All, All for One: A Heterogeneous Data Plane for Flexible P4 Processing, in: IEEE International Conference on Network Protocols (ICNP), 2018, pp. 440–441.

[171] C. Beckmann, R. Krishnamoorthy, H. Wang, A. Lam, C. Kim, Hurdles for a DRAM-based Match-Action Table, in: Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), 2020, pp. 13–16.

[172] A. Aghdai, Y. Xu, H. J. Chao, Design of a hybrid modular switch, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2017, pp. 1–6.

[173] S. Laki, D. Horpacsi, P. Voros, M. Tejfel, P. Hudoba, G. Pongracz, L. Molnar, The Price for Asynchronous Execution of Extern Functions in Programmable Software Data Planes, in: Workshop on Flexible Network Data Plane Processing (NETPROC@ICIN), 2020, pp. 23–28.

[174] D. Horpácsi, P. Vörös, M. Tejfel, S. Laki, G. Pongrácz, L. Molnár, Asynchronous Extern Functions in Programmable Software Data Planes, in: P4 Workshop in Europe (EuroP4), 2019, pp. 1–2.

[175] D. Scholz, A. Oeldemann, F. Geyer, S. Gallenmüller, H. Stubbe, T. Wild, A. Herkersdorf, G. Carle, Cryptographic Hashing in P4 Data Planes, in: P4 Workshop in Europe (EuroP4), 2019, pp. 1–6.

[176] J. S. da Silva, F.-R. Boyer, L.-O. Chiquette, J. P. Langlois, Extern Objects in P4: an ROHC Header Compression Scheme Case Study, in: IEEE Conference on Network Softwarization (NetSoft), 2018, pp. 517–522.

[177] N. Gray, A. Grigorjew, T. Hosssfeld, A. Shukla, T. Zinner, Highlighting the Gap Between Expected and Actual Behavior in P4-enabled Networks, in: IFIP/IEEE Symposium on Integrated Management (IM), 2019, pp. 731–732.

[178] M. V. Dumitru, D. Dumitrescu, C. Raiciu, Can We Exploit Buggy P4 Programs?, in: ACM Symposium on SDN Research (SOSR), 2020, p. 62–68.

[179] J. Mambretti, J. Chen, F. Yeh, S. Y. Yu, International P4 Networking Testbed, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2019, pp. 1–2.

[180] B. Chung, C. Tseng, J. H. Chen, J. Mambretti, P4MT: Multi-Tenant Support Prototype for International P4 Testbed, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2019, pp. 1–2.

[181] A national programmable infrastructure to experiment with next-generation networks, https://www.2stic.nl/national-programmable-infrastructure.html, accessed 01-20-2021 (2021).

[182] R. Sukapuram, G. Barua, PPCU: Proportional Per-packet Consistent Updates for SDNs using Data Plane Time Stamps, Computer Networks 155 (2019) 72–86.

[183] R. Sukapuram, G. Barua, ProFlow: Proportional Per-Bidirectional-Flow Consistent Updates, IEEE Transactions on Network and Service Management (TNSM) 16 (2019) 675–689.

[184] S. Liu, T. A. Benson, M. K. Reiter, Efficient and Safe Network Updates with Suffix Causal Consistency, in: European Conference on Computer Systems (EUROSYS), 2019, p. 1–15.

[185] T. D. Nguyen, M. Chiesa, M. Canini, Decentralized Consistent Network Updates in SDN with ez-Segway, ArXiv e-prints (2017).

[186] S. Geissler, S. Herrnleben, R. Bauer, A. Grigorjew, T. Zinner, M. Jarschel, The Power of Composition: Abstracting aMulti-Device SDN Data Path Through a Single API, IEEE Transactions on Network and Service Management (TNSM) (2019) 722–735.

[187] E. C. Molero, S. Vissicchio, L. Vanbever, Hardware-Accelerated Network Control Planes, in: ACM Workshop on Hot Topics in Networks (HotNets), 2018, p. 120–126.

[188] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, J. Rexford, Heavy-Hitter Detection Entirely in the Data Plane, in: ACM Symposium on SDN Research (SOSR), 2017, p. 164–176.

[189] GitHub: Hashpipe, https://github.com/vibhaa/hashpipe, accessed 01-20-2021 (2021).

[190] Y. Lin, C. Huang, S. Tsai, SDN Soft Computing Application for Detecting Heavy Hitters, IEEE Transactions on Industrial Informatics (ToII) 15 (2019) 5690–5699.

[191] D. A. Popescu, G. Antichi, A. W. Moore, Enabling Fast Hierarchical Heavy Hitter Detection using Programmable Data Planes, in: ACM Symposium on SDN Research (SOSR), 2017, p. 191–192.

[192] R. Harrison, Q. Cai, A. Gupta, J. Rexford, Network-Wide Heavy Hitter Detection with Commodity Switches, in: ACM Symposium on SDN Research (SOSR), 2018, pp. 1–7.

[193] J. Kučera, D. A. Popescu, H. Wang, A. Moore, J. Kořenek, G. Antichi, Enabling Event-Triggered Data Plane Monitoring, in: ACM Symposium on SDN Research (SOSR), 2020, p. 14–26.

[194] M. Silva, A. Jacobs, R. Pfitscher, L. Granville, IDEAFIX: Identifying Elephant Flows in P4-Based IXP Networks, in: IEEE Global Communications Conference (GLOBECOM), 2018, pp. 1–6.

[195] B. Turkovic, J. Oostenbrink, F. Kuipers, Detecting Heavy Hitters in the Data-plane, ArXiv e-prints (2019).

[196] D. Ding, M. Savi, G. Antichi, D. Siracusa, An Incrementally-Deployable P4-Enabled Architecture for Network-Wide Heavy-Hitter Detection, IEEE Transactions on Network and Service Management (TNSM) 17 (2020) 75–88.

[197] GitHub: Network-Wide Heavy-Hitter Detection Implementation in P4 Language, `https://github.com/DINGDAMU/Network-wide-heavy-hitter-detection`, accessed 01-20-2021 (2021).

[198] J. Sonchack, A. J. Aviv, E. Keller, J. M. Smith, Turboflow: Information Rich Flow Record Generation on Commodity Switches, in: European Conference on Computer Systems (EUROSYS), 2018, p. 1–16.

[199] GitHub: TurboFlow, `https://github.com/jsonch/TurboFlow`, accessed 01-20-2021 (2021).

[200] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, J. M. Smith, Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow, in: USENIX Annual Technical Conference (ATC), 2018, pp. 823–835.

[201] GitHub: StarFlow, `https://github.com/jsonch/starflow`, accessed 01-25-2021 (2021).

[202] J. Hill, M. Aloserij, P. Grosso, Tracking Network Flows with P4, in: IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS), 2018, pp. 23–32.

[203] L. Castanheira, R. Parizotto, A. E. Schaeffer-Filho, FlowStalker: Comprehensive Traffic Flow Monitoring on the Data Plane using P4, in: IEEE International Conference on Communicaotions (ICC), 2019, pp. 1–6.

[204] R. Parizotto, L. Castanheira, R. H. Ribeiro, L. Zembruzki, A. S. Jacobs, L. Z. Granville, A. Schaeffer-Filho, ShadowFS: Speeding-up Data Plane Monitoring and Telemetry using P4, in: IEEE International Conference on Communicaotions (ICC), 2020, pp. 1–6.

[205] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, A. Madeira, FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications, in: Network and Distributed Systems Security Symposium (NDSS), 2021, pp. 1–18.

[206] GitHub: FlowLens, https://github.com/dmbb/FlowLens, accessed 04-14-2021 (2021).

[207] W. Wang, P. Tammana, A. Chen, T. S. E. Ng, Grasp the Root Causes in the Data Plane: Diagnosing Latency Problems with SpiderMon, in: ACM Symposium on SDN Research (SOSR), 2020, p. 55–61.

[208] X. Chen, S. Landau-Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, T.-Y. Wang, Fine-Grained Queue Measurement in the Data Plane, in: ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2019, p. 15–29.

[209] Z. Zhao, X. Shi, X. Yin, Z. Wang, Q. Li, HashFlow for Better Flow Record Collection, in: IEEE International Conference on Distributed Computing Systems (ICDCS), 2019, pp. 1416–1425.

[210] Q. Huang, P. P. C. Lee, Y. Bao, Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference, in: ACM SIGCOMM Conference, 2018, p. 576–590.

[211] GitHub: SketchLearn, https://github.com/huangqundl/SketchLearn, accessed 01-20-2021 (2021).

[212] L. Tang, Q. Huang, P. C. Lee, A Fast and Compact Invertible Sketch for Network-Wide Heavy Flow Detection, IEEE/ACM Transactions on Networking (ToN) 28 (2020) 2350–2363.

[213] GitHub: MV-Sketch, https://github.com/Grace-TL/MV-Sketch, accessed 01-20-2021 (2021).

[214] Z. Hang, M. Wen, Y. Shi, C. Zhang, Interleaved Sketch: Toward Consistent Network Telemetry for Commodity Programmable Switches, IEEE ACCESS 7 (2019) 146745–146758.

[215] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, V. Braverman, One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon, in: ACM SIGCOMM Conference, 2016, p. 101–114.

[216] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, S. Uhlig, Elastic Sketch: Adaptive and Fast Network-wide Measurements, in: ACM SIGCOMM Conference, 2018, p. 561–575.

[217] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, S. Uhlig, Adaptive Measurements Using One Elastic Sketch, IEEE/ACM Transactions on Networking (ToN) 27 (2019) 2236–2251.

[218] GitHub: ElasticSketch, https://github.com/BlockLiu/ElasticSketchCode, accessed 01-20-2021 (2021).

[219] F. Pereira, N. Neves, F. M. V. Ramos, Secure network monitoring using programmable data planes, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2017, pp. 286–291.

[220] R. F. T. Martins, F. L. Verdi, R. Villaça, L. F. U. Garcia, Using Probabilistic Data Structures for Monitoring of Multi-tenant P4-based Networks, in: IEEE Symposium on Computers and Communications (ISCC), 2018, pp. 204–207.

[221] Y.-K. Lai, K.-Y. Shih, P.-Y. Huang, H.-P. Lee, Y.-J. Lin, T.-L. Liu, J. H. Chen, Sketch-based Entropy Estimation for Network Traffic Analysis using Programmable Data Plane ASICs, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2019, pp. 1–2.

[222] Z. Liu, S. Zhou, O. Rottenstreich, V. Braverman, J. Rexford, Memory-Efficient Performance Monitoring on Programmable Switches with Lean Algorithms, in: SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS), 2020, pp. 31–44.

[223] L. Tang, Q. Huang, P. P. C. Lee, SpreadSketch: Toward Invertible and Network-Wide Detection of Superspreaders, in: IEEE International Conference on Computer Communications (INFOCOM), 2020, pp. 1608–1617.

[224] GitHub: SpreadSketch, `http://adslab.cse.cuhk.edu.hk/software/spreadsketch/`, accessed 01-20-2021 (2021).

[225] J. Vestin, A. Kassler, D. Bhamare, K. Grinnemo, J. Andersson, G. Pongracz, Programmable Event Detection for In-Band Network Telemetry, in: IEEE International Conference on Cloud Networking (IEEE CloudNet), 2019, pp. 1–6.

[226] S. Wang, Y. Chen, J. Li, H. Hu, J. Tsai, Y. Lin, A Bandwidth-Efficient INT System for Tracking the Rules Matched by the Packets of a Flow, in: IEEE Global Communications Conference (GLOBECOM), 2019, pp. 1–6.

[227] D. Bhamare, A. Kassler, J. Vestin, M. A. Khoshkholghi, J. Taheri, IntOpt: In-Band Network Telemetry Optimization for NFV Service Chain Monitoring, in: IEEE International Conference on Communicaotions (ICC), 2019, pp. 1–7.

[228] C. Jia, T. Pan, Z. Bian, X. Lin, E. Song, C. Xu, T. Huang, Y. Liu, Rapid Detection and Localization of Gray Failures in Data Centers via In-band Network Telemetry, in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2020, pp. 1–9.

[229] GitHub: Gray Failures Detection and Localization, `https://github.com/graytower/INT_DETECT`, accessed 01-20-2021 (2021).

[230] B. Niu, J. Kong, S. Tang, Y. Li, Z. Zhu, Visualize Your IP-Over-Optical Network in Realtime: A P4-Based Flexible Multilayer In-Band Network Telemetry (ML-INT) System, IEEE ACCESS 7 (2019) 82413–82423.

[231] N. S. Kagami, R. I. T. da Costa Filho, L. P. Gaspary, CAPEST: Offloading Network Capacity and Available Bandwidth Estimation to Programmable Data Planes, IEEE Transactions on Network and Service Management (TNSM) 17 (2020) 175–189.

[232] GitHub: Capest, `https://github.com/nicolaskagami/capest`, accessed 01-20-2021 (2021).

[233] N. Choi, L. Jagadeesan, Y. Jin, N. N. Mohanasamy, M. R. Rahman, K. Sabnani, M. Thottan, Run-time Performance Monitoring, Verification, and Healing of End-to-End Services, in: IEEE Conference on Network Softwarization (NetSoft), 2019, pp. 30–35.

[234] A. Sgambelluri, F. Paolucci, A. Giorgetti, D. Scano, F. Cugini, Exploiting Telemetry in Multi-Layer Networks, in: International Conference on Transparent Optical Networks (ICTON), 2020, pp. 1–4.

[235] Y. Feng, S. Panda, S. G. Kulkarni, K. K. Ramakrishnan, N. Duffield, A SmartNIC-Accelerated Monitoring Platform for In-band Network Telemetry, in: IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), 2020, pp. 1–6.

[236] J. Marques, K. Levchenko, L. Gaspary, IntSight: Diagnosing SLO Violations with in-Band Network Telemetry, in: ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2020, p. 421–434.

[237] GitHub: IntSight, `https://github.com/jonadmark/intsight-conext`, accessed 01-20-2021 (2021).

[238] D. Suh, S. Jang, S. Han, S. Pack, X. Wang, Flexible sampling-based in-band network telemetry in programmable data plane, ICT Express 6 (2020) 62–65.

[239] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, C. Kim, Language-Directed Hardware Design for Network Performance Monitoring, in: ACM SIGCOMM Conference, 2017, p. 85–98.

[240] V. Nathan, S. Narayana, A. Sivaraman, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, C. Kim, Demonstration of the Marple System for Network Performance Monitoring, in: ACM SIGCOMM Conference Posters and Demos, 2017, p. 57–59.

[241] GitHub: Marple, `https://github.com/performance-queries/marple`, accessed 01-20-2021 (2021).

[242] P. Laffranchini, L. Rodrigues, M. Canini, B. Krishnamurthy, Measurements As First-class Artifacts, in: IEEE International Conference on Computer Communications (INFOCOM), 2019, pp. 415–423.

[243] GitHub: Mafia, `https://github.com/paololaff/mafia-sdn`, accessed 01-20-2021 (2021).

[244] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, W. Willinger, Sonata: Query-Driven Streaming Network Telemetry, in: ACM Symposium on SDN Research (SOSR), 2018, p. 357–371.

[245] GitHub: SONATA, `https://github.com/Sonata-Princeton/SONATA-DEV`, accessed 01-20-2021 (2021).

[246] R. Teixeira, R. Harrison, A. Gupta, J. Rexford, PacketScope: Monitoring the Packet Lifecycle Inside a Switch, in: ACM Symposium on SDN Research (SOSR), 2020, p. 76–82.

[247] Y. Gao, Y. Jing, W. Dong, UniROPE: Universal and Robust Packet Trajectory Tracing for Software-Defined Networks, IEEE/ACM Transactions on Networking (ToN) 26 (2018) 2515–2527.

[248] S. Knossen, J. Hill, P. Grosso, Hop Recording and Forwarding State Logging: Two Implementations for Path Tracking in P4, in: IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS), 2019, pp. 36–47.

[249] A. Indra Basuki, D. Rosiyadi, I. Setiawan, Preserving Network Privacy on Fine-grain Path-tracking Using P4-based SDN, in: International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET), 2020, pp. 129–134.

[250] R. Joshi, T. Qu, M. C. Chan, B. Leong, B. T. Loo, BurstRadar: Practical Real-time Microburst Monitoring for Datacenter Networks, in: ACM SIGOPS Asia-Pacific Workshop on System (APSys), 2018, pp. 1–8.

[251] GitHub: BurstRadar, `https://github.com/harshgondaliya/burstradar`, accessed 01-20-2021 (2021).

[252] M. Ghasemi, T. Benson, J. Rexford, Dapper: Data Plane Performance Diagnosis of TCP, in: ACM Symposium on SDN Research (SOSR), 2017, p. 61–74.

[253] C.-H. He, B. Y. Chang, S. Chakraborty, C. Chen, L. C. Wang, A Zero Flow Entry Expiration Timeout P4 Switch, in: ACM Symposium on SDN Research (SOSR), 2018, pp. 1–2.

[254] A. Riesenberg, Y. Kirzon, M. Bunin, E. Galili, G. Navon, T. Mizrahi, Time-Multiplexed Parsing in Marking-Based Network Telemetry, in: ACM International Conference on Systems and Storage (SYSTOR), 2019, p. 80–85.

[255] GitHub: P4 Alternate Marking Algorithm, `https://github.com/AlternateMarkingP4/FlaseClase`, accessed 01-20-2021 (2021).

[256] S. Y. Wang, H. W. Hu, Y. B. Lin, Design and Implementation of TCP-Friendly Meters in P4 Switches, IEEE/ACM Transactions on Networking (ToN) 28 (2020) 1885–1898.

[257] R. Kundel, F. Siegmund, J. Blendin, A. Rizk, B. Koldehofe, P4STA: High Performance Packet Timestamping with Programmable Packet Processors, in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2020, p. 1–9.

[258] GitHub: P4STA, `https://github.com/ralfkundel/P4STA`, accessed 01-20-2021 (2021).

[259] R. Hark, D. Bhat, M. Zink, R. Steinmetz, A. Rizk, Preprocessing Monitoring Information on the SDN Data-Plane using P4, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2019, pp. 1–6.

[260] D. Ding, M. Savi, D. Siracusa, Estimating Logarithmic and Exponential Functions to Track Network Traffic Entropy in P4, in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2020, pp. 1–9.

[261] GitHub: P4Entropy, `https://github.com/DINGDAMU/P4Entropy`, accessed 01-20-2021 (2021).

[262] P. Taffet, J. Mellor-Crummey, Lightweight, Packet-Centric Monitoring of Network Traffic and Congestion Implemented in P4, in: IEEE Symposium on High-Performance Interconnects (HOTI), 2019, pp. 54–58.

[263] Y. Lin, Y. Zhou, Z. Liu, K. Liu, Y. Wang, M. Xu, J. Bi, Y. Liu, J. Wu, NetView: Towards On-Demand Network-Wide Telemetry in the Data Center, in: IEEE International Conference on Communicaotions (ICC), 2020, pp. 1–6.

[264] J. Bai, M. Zhang, G. Li, C. Liu, M. Xu, H. Hu, FastFE: Accelerating ML-Based Traffic Analysis with Programmable Switches, in: Workshop on Secure Programmable Network Infrastructure (SPIN), 2020, p. 1–7.

[265] J. Kučera, R. B. Basat, M. Kuka, G. Antichi, M. Yu, M. Mitzenmacher, Detecting Routing Loops in the Data Plane, in: ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2020, p. 466–473.

[266] Z. Hang, Y. Shi, M. Wen, C. Zhang, TBSW: Time-Based Sliding Window Algorithm for Network Traffic Measurement, in: IEEE International Conference on High Performance Computing and Communications; IEEE International Conference on Smart City; IEEE International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2019, pp. 1305–1310.

[267] B. Guan, S. Shen, FlowSpy: An Efficient Network Monitoring Framework Using P4 in Software-Defined Networks, in: IEEE Semiannual Vehicular Technology Conference (VTC), 2019, pp. 1–5.

[268] Heavy Hitter Detection: Guest lecture for CS344 at Stanford, `https://cs344-stanford.github.io/lectures/Lecture-4-HHD.pdf`, accessed 01-20-2021 (2018).

[269] B. Claise, Cisco Systems NetFlow Services Export Version 9, RFC 3954, RFC Editor (10 2004).
URL `http://www.rfc-editor.org/rfc/rfc3954.txt`

[270] P. Phaal, S. Panchen, N. McKee, InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks, RFC 3176, RFC Editor (09 2001).
URL `http://www.rfc-editor.org/rfc/rfc3176.txt`

[271] B. Claise, B. Trammell, P. Aitken, Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information, STD 77, RFC Editor (09 2013).
URL `http://www.rfc-editor.org/rfc/rfc7011.txt`

[272] In-band Network Telemetry (INT), `https://p4.org/assets/INT-current-spec.pdf`, accessed 01-20-2021 (2021).

[273] Charter of the P4 Applications WG, `https://github.com/p4lang/p4-applications/blob/master/docs/charter.pdf`, accessed 01-20-2021 (2021).

[274] C. Kim, A. Sivaraman, N. P. Katta, A. Bas, A. Dixit, L. J. Wobker, In-band Network Telemetry via Programmable Dataplanes, `https://nkatta.github.io/papers/int-demo.pdf` (2015).

[275] F. Cugini, P. Gunning, F. Paolucci, P. Castoldi, A. Lord, P4 In-Band Telemetry (INT) for Latency-Aware VNF in Metro Networks, in: Optical Fiber Communication Conference (OFC), 2019, pp. 1–3.

[276] Open Networking Foundation: Trellis, `https://www.opennetworking.org/trellis/`, accessed 01-20-2021 (2021).

[277] Google Presentations: Trellis & P4 Tutorial, `http://bit.ly/trellis-p4-slides`, accessed 01-20-2021 (2018).

[278] GitHub: ONF Trellis, `https://github.com/opennetworkinglab/routing/tree/master/trellis`, accessed 01-20-2021 (2021).

[279] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, M. Budiu, DC.P4: Programming the Forwarding Plane of a Data-center Switch, in: ACM SIGCOMM Conference, 2015, p. 1–8.

[280] GitHub: DC.p4, `https://github.com/p4lang/papers/tree/master/sosr15`, accessed 01-20-2021 (2021).

[281] Open Network Foundation: P4 apps at ONF, `https://github.com/p4lang/p4-applications/blob/master/meeting_slides/2018_04_19_ONF.pdf`, accessed 01-20-2021 (2018).

[282] GitHub: fabric.p4, `https://github.com/opennetworkinglab/onos/blob/master/pipelines/fabric/impl/src/main/resources/fabric.p4`, accessed 01-20-2021 (2021).

[283] RARE (Router for Academia, Research & Education), https://wiki.geant.org/display/RARE/Home, accessed 04-16-2021 (2021).

[284] GitHub: RARE, `https://github.com/frederic-loui/RARE`, accessed 04-16-2021 (2021).

[285] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, M. Townsley, T. Clausen, Stateless Load-Aware Load Balancing in P4, in: IEEE International Conference on Network Protocols (ICNP), 2018, pp. 418–423.

[286] R. Miao, H. Zeng, C. Kim, J. Lee, M. Yu, SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap using Switching ASICs, in: ACM SIGCOMM Conference, 2017, p. 15–28.

[287] N. Katta, M. Hira, C. Kim, A. Sivaraman, J. Rexford, HULA: Scalable Load Balancing using Programmable Data Planes, in: ACM Symposium on SDN Research (SOSR), 2016, p. 1–12.

[288] C. H. Benet, A. J. Kassler, T. Benson, G. Pongracz, MP-HULA: Multipath Transport Aware Load Balancing using Programmable Data Planes, in: Morning Workshop on In-Network Computing, 2018, p. 7–13.

[289] B. T. Chiang, K. Wang, Cost-effective Congestion-aware Load Balancing for Datacenters, in: International Conference on Electronics, Information, and Communication (ICEIC), 2019, pp. 1–6.

[290] J.-L. Ye, C. Chen, Y. H. Chu, A Weighted ECMP Load Balancing Scheme for Data Centers using P4 Switches, in: IEEE International Conference on Cloud Networking (IEEE CloudNet), 2018, pp. 1–4.

[291] K.-F. Hsu, P. Tammana, R. Beckett, A. Chen, J. Rexford, D. Walker, Adaptive Weighted Traffic Splitting in Programmable Data Planes, in: ACM Symposium on SDN Research (SOSR), 2020, p. 103–109.

[292] M. Pizzutti, A. Schaeffer-Filho, An Efficient Multipath Mechanism Based on the Flowlet Abstraction and P4, in: IEEE Global Communications Conference (GLOBECOM), 2018, pp. 1–6.

[293] M. Pizzutti, A. Schaeffer-Filho, Adaptive Multipath Routing based on Hybrid Data and Control Plane Operation, in: IEEE International Conference on Computer Communications (INFOCOM), 2020, pp. 730–738.

[294] J. Zhang, S. Wen, J. Zhang, H. Chai, T. Pan, T. Huang, L. Zhang, Y. Liu, F. R. Yu, Fast Switch-Based Load Balancer Considering Application Server States, IEEE/ACM Transactions on Networking (ToN) 28 (2020) 1391–1404.

[295] Q. Li, J. Zhang, T. Pan, T. Huang, Y. Liu, Data-driven Routing Optimization based on Programmable Data Plane, in: IEEE International Conference on Computer Communications and Networks (ICCCN), 2020, pp. 1–9.

[296] E. Kawaguchi, H. Kasuga, N. Shinomiya, Unsplittable flow Edge Load factor Balancing in SDN using P4 Runtime, in: International Telecommunication Networks and Applications Conference (ITNAC), 2019, pp. 1–6.

[297] E. Cidon, S. Choi, S. Katti, N. McKeown, AppSwitch: Application-layer Load Balancing withina Software Switch, in: Asia-Pacific Workshop on Networking (APnet), 2017, p. 64–70.

[298] V. Olteanu, A. Agache, A. Voinescu, C. Raiciu, Stateless Datacenter Load-balancing with Beamer, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2018, pp. 125–139.

[299] GitHub: Beamer, `https://github.com/Beamer-LB`, accessed 01-25-2021 (2021).

[300] J. Geng, J. Yan, Y. Zhang, P4QCN: Congestion Control using P4-Capable Device in Data Center Networks, Electronics Journal 8 (2019) 280.

[301] J. Jiang, Y. Zhang, An Accurate Congestion Control Mechanism in Programmable Network, in: IEEE Annual Computing and Communication Workshop and Conference (CCWC), 2019, pp. 673–677.

[302] S. Shahzad, E. Jung, J. Chung, R. Kettimuthu, Enhanced Explicit Congestion Notification (EECN) in TCP with P4 Programming, in: International Conference on Green and Human Information Technology (ICGHIT), 2020, pp. 35–40.

[303] C. Chen, H. Fang, M. S. Iqbal, QoSTCP: Provide Consistent Rate Guarantees to TCP flows in Software Defined Networks, in: IEEE International Conference on Communicaotions (ICC), 2020, pp. 1–6.

[304] A. Laraba, J. François, I. Chrisment, S. R. Chowdhury, R. Boutaba, Defeating Protocol Abuse with P4: Application to Explicit Congestion Notification, in: IFIP-TC6 Networking Conference (Networking), 2020, pp. 431–439.

[305] N. K. Sharma, M. Liu, K. Atreya, A. Krishnamurthy, Approximating Fair Queueing on Reconfigurable Switches, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2018, p. 1–16.

[306] C. Cascone, N. Bonelli, L. Bianchi, A. Capone, B. Sansò, Towards Approximate Fair Bandwidth Sharing via Dynamic Priority Queuing, in: IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), 2017, pp. 1–6.

[307] D. Bhat, J. Anderson, P. Ruth, M. Zink, K. Keahey, Application-based QoE support with P4 and OpenFlow, in: IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2019, pp. 817–823.

[308] E. F. Kfoury, J. Crichigno, E. Bou-Harb, D. Khoury, G. Srivastava, Enabling TCP Pacing using Programmable Data Plane Switches, in: International Conference on Telecommunications and Signal Processing (TSP), 2019, pp. 273–277.

[309] Y. Chen, L. Yen, W. Wang, C. Chuang, Y. Liu, C. Tseng, P4-Enabled Bandwidth Management, in: Asia-Pacific Network Operations and Management Symposium (APNOMS), 2019, pp. 1–5.

[310] S. S. W. Lee, K. Chan, A Traffic Meter Based on a Multicolor Marker for Bandwidth Guarantee and Priority Differentiation in SDN Virtual Networks, IEEE Transactions on Network and Service Management (TNSM) 16 (2019) 1046–1058.

[311] S.-Y. Wang, J.-Y. Li, Y.-B. Lin, Aggregating and disaggregating packets with various sizes of payload in P4 switches at 100 Gbps line rate, Journal of Network and Computer Applications (JNCA) 165 (2020) 102676.

[312] K. Tokmakov, M. Sarker, J. Domaschka, S. Wesner, A Case for Data Centre Traffic Management on Software Programmable Ethernet Switches, in: IEEE International Conference on Cloud Networking (IEEE CloudNet), 2019, pp. 1–6.

[313] B. Turkovic, F. Kuipers, N. van Adrichem, K. Langendoen, Fast Network Congestion Detection and Avoidance using P4, in: Workshop on Networking for Emerging Applications and Technologies (NEAT), 2018, p. 45–51.

[314] R. Kundel, J. Blendin, T. Viernickel, B. Koldehofe, R. Steinmetz, P4-CoDel: Active Queue Management in Programmable Data Planes, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2018, pp. 1–4.

[315] GitHub: P4-CoDel, `https://github.com/ralfkundel/p4-codel`, accessed 01-20-2021 (2021).

[316] M. Menth, H. Mostafaei, D. Merling, M. Häberle, Implementation and Evaluation of Activity-Based Congestion Management using P4 (P4-ABC), MDPI Future Internet Journal (FI) 11 (2019) 159.

[317] B. Turkovic, F. Kuipers, P4air: Increasing Fairness among Competing Congestion Control Algorithms, in: IEEE International Conference on Network Protocols (ICNP), 2020, pp. 1–12.

[318] L. B. Fernandes, L. Camargos, Bandwidth throttling in a P4 switch, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2020, pp. 91–94.

[319] G. Wang, C. Chen, C. Chen, L. Pan, Y. Wang, C. Fan, C. Hsu, Streaming Scalable Video Sequences with Media-Aware Network Elements Implemented in P4 Programming Language, in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2018, pp. 1–2.

[320] A. G. Alcoz, A. Dietmüller, L. Vanbever, SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2020, pp. 59–76.

[321] I. Kunze, M. Gunz, D. Saam, K. Wehrle, J. Rüth, Tofino + P4: A Strong Compound for AQM on High-Speed Networks?, in: IFIP/IEEE International Symposium on Integrated Network Management, 2021, pp. 72–80.

[322] GitHub: PIE for Tofino, `https://github.com/COMSYS/pie-for-tofino`, accessed 04-15-2021 (2021).

[323] H. Harkous, C. Papagianni, K. De Schepper, M. Jarschel, M. Dimolianis, R. Preis, Virtual queues for p4: A poor man's programmable traffic manager, IEEE Transactions on Network and Service Management (TNSM) (2021) 1–1.

[324] B. Andrus, S. A. Sasu, T. Szyrkowiec, A. Autenrieth, M. Chamania, J. K. Fischer, S. Rasp, Zero-Touch Provisioning of Distributed Video Analytics in a Software-Defined Metro-Haul Network with P4 Processing, in: Optical Fiber Communication Conference (OFC), 2019, pp. 1–3.

[325] S. Ibanez, G. Antichi, G. Brebner, N. McKeown, Event-Driven Packet Processing, in: ACM Workshop on Hot Topics in Networks (HotNets), 2019, p. 133–140.

[326] E. F. Kfoury, J. Crichigno, E. Bou-Harb, Offloading Media Traffic to Programmable Data Plane Switches, in: IEEE International Conference on Communicaotions (ICC), 2020, pp. 1–7.

[327] I. Kettaneh, S. Udayashankar, A. Abdel-hadi, R. Grosman, S. Al-Kiswany, Falcon: Low Latency, Network-Accelerated Scheduling, in: P4 Workshop in Europe (EuroP4), 2020, p. 7–12.

[328] T. Osiński, M. Kossakowski, M. Pawlik, J. Palimąka, M. Sala, H. Tarasiuk, Unleashing the Performance of Virtual BNG by Offloading Data Plane to a Programmable ASIC, in: P4 Workshop in Europe (EuroP4), 2020, p. 54–55.

[329] J. Lee, R. Miao, C. Kim, M. Yu, H. Zeng, Stateful Layer-4 Load Balancing in Switching ASICs, in: ACM SIGCOMM Conference Posters and Demos, 2017, p. 133–135.

[330] K. Nichols, V. Jacobson, A. McGregor, J. Iyengar, Controlled Delay Active Queue Management, RFC 8289, RFC Editor (01 2018).
URL `https://tools.ietf.org/rfc/rfc8289.txt`

[331] B. Lewis, L. Fawcett, M. Broadbent, N. Race, Using P4 to Enable Scalable Intents in Software Defined Networks, in: IEEE International Conference on Network Protocols (ICNP), 2018, pp. 442–443.

[332] GitHub: P4 Source Routing, `https://github.com/BenRLewis/P4-Source-Routing`, accessed 01-20-2021 (2021).

[333] L. Luo, H. Yu, S. Luo, Z. Ye, X. Du, M. Guizani, Scalable Explicit Path Control in Software-Defined Networks, Journal of Network and Computer Applications (JNCA) 141 (2019) 86–103.

[334] GitHub: P4 Paco, `https://github.com/an15m/paco`, accessed 01-20-2021 (2021).

[335] A. Kushwaha, S. Sharma, N. Bazard, A. Gumaste, B. Mukherjee, Design, Analysis, and a Terabit Implementation of a Source-Routing-Based SDN Data Plane, IEEE Systems Journal (2020).

[336] A. Abdelsalam, A. Tulumello, M. Bonola, S. Salsano, C. Filsfils, Pushing Network Programmability to the limits with SRv6 uSIDs and P4, in: P4 Workshop in Europe (EuroP4), 2020, p. 62–64.

[337] W. Braun, J. Hartmann, M. Menth, Demo: Scalable and Reliable Software-Defined Multicast with BIER and P4, in: IFIP/IEEE Symposium on Integrated Management (IM), 2017, pp. 905–906.

[338] Bitbucket: p4-bfr), `https://bitbucket.org/wb-ut/p4-bfr`, accessed 01-20-2021 (2021).

[339] D. Merling, S. Lindner, M. Menth, P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast, Journal of Network and Computer Applications (JNCA) 169 (2020) 102764.

[340] D. Merling, S. Lindner, M. Menth, Hardware-based evaluation of scalable and resilient multicast with bier in p4, IEEE ACCESS 9 (2021) 34500–34514.

[341] GitHub: P4-BIER, `https://github.com/uni-tue-kn/p4-bier`, accessed 01-20-2021 (2021).

[342] GitHub: P4-BIER for Tofino, `https://github.com/uni-tue-kn/p4-bier-tofino`, accessed 04-26-2021 (2021).

[343] M. Shahbaz, L. Suresh, J. Rexford, N. Feamster, O. Rottenstreich, M. Hira, Elmo: Source Routed Multicast for Public Clouds, in: ACM Special Interest Group on Data Communication, 2019, pp. 2587–2600.

[344] GitHub: Elmo MCast, `https://github.com/Elmo-MCast/p4-programs`, accessed 01-20-2021 (2021).

[345] S. Luo, H. Yu, K. Li, H. Xing, Efficient File Dissemination in Data Center Networks with Priority-based Adaptive Multicast, IEEE Journal on Selected Areas in Communications (JSAC) 38 (2020) 1161–1175.

[346] C. Wernecke, H. Parzyjegla, G. Mühl, P. Danielis, D. Timmermann, Realizing Content-Based Publish/Subscribe with P4, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2018, pp. 1–7.

[347] C. Wernecke, H. Parzyjegla, G. Mühl, E. Schweissguth, D. Timmermann, Flexible Notification Forwarding for Content-Based Publish/Subscribe Using P4, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2020, pp. 1–5.

[348] C. Wernecke, H. Parzyjegla, G. Mühl, Implementing Content-based Publish/Subscribe on the Network Layer with P4, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2020, pp. 144–149.

[349] C. Wernecke, H. Parzyjegla, G. Mühl, P. Danielis, E. Schweissguth, D. Timmermann, Stitching Notification Distribution Trees for Content-based Publish/Subscribe with P4, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2020, pp. 100–104.

[350] T. Jepsen, M. Moshref, A. Carzaniga, N. Foster, R. Soulé, Packet Subscriptions for Programmable ASICs, in: ACM Workshop on Hot Topics in Networks (HotNets), 2018, p. 176–183.

[351] R. Kundel, C. Gaertner, M. Luthra, S. Bhowmik, B. Koldehofe, Flexible Content-based Publish/Subscribe over Programmable Data Planes, in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2020, pp. 1–5.

[352] GitHub: p4bsub, `https://github.com/ralfkundel/p4bsub/`, accessed 01-20-2021 (2021).

[353] J. Vestin, A. Kassler, S. Laki, G. Pongrácz, Towards In-Network Event Detection and Filtering for Publish/Subscribe Communication using Programmable Data Planes, IEEE Transactions on Network and Service Management (TNSM) (2020) 415–428.

[354] S. Signorello, R. State, J. François, O. Festor, NDN.p4: Programming Information-Centric Data-Planes, in: IEEE Conference on Network Softwarization (NetSoft), 2016, pp. 384–389.

[355] R. Miguel, S. Signorello, F. M. V. Ramos, Named Data Networking with Programmable Switches, in: IEEE International Conference on Network Protocols (ICNP), 2018, pp. 400–405.

[356] GitHub: NDN.p4, `https://github.com/signorello/NDN.p4`, accessed 01-20-2021 (2021).

[357] GitHub: NDN.p4-16, `https://github.com/netx-ulx/NDN.p4-16`, accessed 01-20-2021 (2021).

[358] O. Karrakchou, N. Samaan, A. Karmouch, ENDN: An Enhanced NDN Architecture with a P4-programmable Data Plane, in: International Conference on Networking (ICN), 2020, p. 1–11.

[359] R. Sedar, M. Borokhovich, M. Chiesa, G. Antichi, S. Schmid, Supporting Emerging Applications With Low-Latency Failover in P4, in: Workshop on Networking for Emerging Applications and Technologies (NEAT), 2018, p. 52–57.

[360] GitHub: P4-FRR, `https://bitbucket.org/roshanms/p4-frr/src/master/`, accessed 01-20-2021 (2021).

[361] H. Giesen, L. Shi, J. Sonchack, A. Chelluri, N. Prabhu, N. Sultana, L. Kant, A. J. McAuley, A. Poylisher, A. DeHon, B. T. Loo, In-Network Computing to the Rescue of Faulty Links, in: Morning Workshop on In-Network Computing, 2018, pp. 1–6.

[362] T. Qu, R. Joshi, M. Chan, B. Leong, D. Guo, Z. Liu, SQR: In-network Packet Loss Recovery from Link Failures for Highly Reliable Datacenter Networks, in: IEEE International Conference on Network Protocols (ICNP), 2019, pp. 1–12.

[363] GitHub: P4 SQR, `https://git.io/fjbnV`, accessed 01-20-2021 (2021).

[364] S. Lindner, D. Merling, M. Häberle, M. Menth, P4-Protect: 1+1 Path Protection for P4, in: P4 Workshop in Europe (EuroP4), 2020, p. 21–27.

[365] GitHub: P4-Protect BMv2, `https://github.com/uni-tue-kn/p4-protect`, accessed 01-20-2021 (2021).

[366] GitHub: P4-Protect Tofino, `https://github.com/uni-tue-kn/p4-protect-tofino`, accessed 01-20-2021 (2021).

[367] K. Hirata, , T. Tachibana, Implementation of Multiple Routing Configurations on Software-Defined Networks with P4, in: Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), 2019, pp. 13–16.

[368] S. Lindner, M. Häberle, F. Heimgaertner, N. Nayak, S. Schildt, D. Grewe, H.Loehr, M. Ment, P4 In-Network Source Protection for Sensor Failover, in: IFIP-TC6 Networking Conference (Networking), 2020, pp. 791–796.

[369] GitHub: P4 Source Protection BMv2, `https://github.com/uni-tue-kn/p4-source-protection`, accessed 01-20-2021 (2021).

[370] GitHub: P4 Source Protection Tofino, `https://github.com/uni-tue-kn/p4-source-protection-tofino`, accessed 01-20-2021 (2021).

[371] K. Subramanian, A. Abhashkumar, L. D'Antoni, A. Akella, D2R: Dataplane-Only Policy-Compliant Routing Under Failures (2019).

[372] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisiński, G. Nikolaidis, S. Schmid, PURR: A Primitive for Reconfigurable Fast Reroute, in: ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2019, p. 1–14.

[373] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, L. Vanbever, Blink: Fast Connectivity Recovery Entirely in the Data Plane, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2019, pp. 161–176.

[374] GitHub: Blink, `https://github.com/nsg-ethz/Blink`, accessed 01-20-2021 (2021).

[375] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, D. Walker, Contra: A Programmable System for Performance-aware Routing, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2020, pp. 701–721.

[376] O. Michel, E. Keller, Policy Routing using Process-Level Identifiers, in: IEEE International Conference on Cloud Engineering Workshop (IC2EW), 2016, pp. 7–12.

[377] A. C. Baktir, A. Ozgovde, C. Ersoy, Implementing Service-Centric Model with P4: A Fully-Programmable Approach, in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2018, pp. 1–6.

[378] W. Froes, L. Santos, L. N. Sampaio, M. Martinello, A. Liberato, R. S. Villaca, ProgLab: Programmable Labels for QoS Provisioning on Software Defined Networks, Computer Communications 161 (2020) 99–108.

[379] N. VARYANI, Z.-L. ZHANG, D. DAI, QROUTE: An Efficient Quality of Service (QoS) Routing Scheme for Software-Defined Overlay Networks, IEEE ACCESS 8 (2020) 104109–104126.

[380] S. Gimenez, E. Grasa, S. Bunch, A Proof of Concept Implementation of a RINA Interior Router using P4-enabled Software Targets, in: Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), 2020, pp. 57–62.

[381] W. Feng, X. Tan, Y. Jin, Implementing ICN over P4 in HTTP Scenario, in: IEEE International Conference on Hot Information-Centric Networking (HotICN), 2019, pp. 37–43.

[382] G. Grigoryan, Y. Liu, M. Kwon, PFCA: A Programmable FIB Caching Architecture, IEEE/ACM Transactions on Networking (ToN) 28 (2020) 1872–1884.

[383] A. McAuley, Y. M. Gottlieb, L. Kant, J. Lee, A. Poylisher, P4-Based Hybrid Error Control Booster Providing New Design Tradeoffs in Wireless Networks, in: IEEE Military Communications Conference (MILCOM), 2019, pp. 731–736.

[384] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, E. Bugnion, R2P2: Making RPCs first-class datacenter citizens, in: USENIX Annual Technical Conference (ATC), 2019, pp. 863–880.

[385] GitHub: R2P2 - Request Response Pair Protocol, `https://github.com/epfl-dcsl/r2p2`, accessed 01-25-2021 (2021).

[386] D. Merling, M. Menth, N. Warnke, T. Eckert, An Overview of Bit Index Explicit Replication (BIER), IETF Journal (2018).

[387] M. Hollingsworth, J. Lee, Z. Liu, J. Lee, S. Ha, D. Grunwald, P4EC: Enabling Terabit Edge Computing in Enterprise 4G LTE, in: USENIX Workshop on Hot Topics in Edge Computing (HotEdge), 2020, pp. 1–7.

[388] GitHub: spgw.p4, `https://github.com/opennetworkinglab/onos/blob/master/pipelines/fabric/impl/src/main/resources/include/control/spgw.p4`, accessed 01-20-2021 (2021).

[389] P. Palagummi, K. M. Sivalingam, SMARTHO: A Network Initiated Handover in NG-RAN using P4-based Switches, in: International Conference on Network and Services Management (CNSM), 2018, pp. 338–342.

[390] A. Aghdai, M. Huang, D. Dai, Y. Xu, J. Chao, Transparent Edge Gateway for Mobile Networks, in: IEEE International Conference on Network Protocols (ICNP), 2018, pp. 412–417.

[391] A. Aghdai, Y. Xu, M. Huang, D. H. Dai, H. J. Chao, Enabling Mobility in LTE-Compatible Mobile-edge Computing with Programmable Switches, ArXiv e-prints (2019).

[392] J. Xie, C. Qian, D. Guo, X. Li, S. Shi, H. Chen, Efficient Data Placement and Retrieval Services in Edge Computing, in: IEEE International Conference on Distributed Computing Systems (ICDCS), 2019, pp. 1029–1039.

[393] J. Xie, D. Guo, X. Shi, H. Cai, C. Qian, H. Chen, A Fast Hybrid Data Sharing Framework for Hierarchical Mobile Edge Computing, in: IEEE International Conference on Computer Communications (INFOCOM), 2020, pp. 2609–2618.

[394] C. Shen, D. Lee, C. Ku, M. Lin, K. Lu, S. Tan, A Programmable and FPGA-accelerated GTP Offloading Engine for Mobile Edge Computing in 5G Networks, in: IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2019, pp. 1021–1022.

[395] C. Lee, K. Ebisawa, H. Kuwata, M. Kohno, S. Matsushima, Performance Evaluation of GTP-U and SRv6 Stateless Translation, in: International Conference on Network and Services Management (CNSM), 2019, pp. 1–6.

[396] R. Ricart-Sanchez, P. Malagon, J. M. Alcaraz-Calero, Q. Wang, P4-NetFPGA-based network slicing solution for 5G MEC architectures, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2019, pp. 1–2.

[397] S. K. Singh, C. E. Rothenberg, G. Patra, G. Pongracz, Offloading Virtual Evolved Packet Gateway User Plane Functions to a Programmable ASIC, in: ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms, 2019, p. 9–14.

[398] R. Shah, V. Kumar, M. Vutukuru, P. Kulkarni, TurboEPC: Leveraging Dataplane Programmability to Accelerate the Mobile Packet Core, in: ACM Symposium on SDN Research (SOSR), 2020, p. 83–95.

[399] P. Vörös, G. Pongrácz, S. Laki, Towards a Hybrid Next Generation NodeB, in: P4 Workshop in Europe (EuroP4), 2020, p. 56–58.

[400] Y. Lin, T. Huang, S. Tsai, Enhancing 5G/IoT Transport Security Through Content Permutation, IEEE ACCESS 7 (2019) 94293–94299.

[401] M. Uddin, S. Mukherjee, H. Chang, T. V. Lakshman, SDN-Based Service Automation for IoT, in: IEEE International Conference on Network Protocols (ICNP), 2017, pp. 1–10.

[402] M. Uddin, S. Mukherjee, H. Chang, T. V. Lakshman, SDN-Based Multi-Protocol Edge Switching for IoT Service Automation, IEEE Journal on Selected Areas in Communications (JSAC) 36 (2018) 2775–2786.

[403] S.-Y. Wang, C.-M. Wu, Y.-B. Linm, C.-C. Huang, High-Speed Data-Plane Packet Aggregation and Disaggregation by P4 Switches, Journal of Network and Computer Applications (JNCA) 142 (2019) 98–110.

[404] A. L. R. Madureira, F. R. C. Araújo, L. N. Sampaio, On supporting IoT data aggregation through programmable data planes, Computer Networks 177 (2020) 107330.

[405] P. Engelhard, A. Zachlod, J. Schulz-Zander, S. Du, Toward scalable and virtualized massive wireless sensor networks, in: International Conference on Networked Systems (NetSys), 2019, pp. 1–6.

[406] J. Vestin, A. Kassler, J. Åkerberg, FastReact: In-Network Control and Caching for Industrial Control Networks using Programmable Data Planes, in: IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2018, pp. 219–226.

[407] F. E. R. Cesen, L. Csikor, C. Recalde, C. E. Rothenberg, G. Pongrácz, Towards Low Latency Industrial Robot Control in Programmable Data Planes, in: IEEE Conference on Network Softwarization (NetSoft), 2020, pp. 165–169.

[408] I. Kunze, R. Glebke, J. Scheiper, M. Bodenbenner, R. H. Schmitt, K. Wehrle, Investigating the Applicability of In-Network Computing to Industrial Scenarios, in: International Conference on Industrial Cyber-Physical Systems (ICPS), 2021, pp. 334–340.

[409] J. Rüth, R. Glebke, K. Wehrle, V. Causevic, S. Hirche, Towards In-Network Industrial Feedback Control, in: Morning Workshop on In-Network Computing, 2018, p. 14–19.

[410] P. G. Kannan, R. Joshi, M. C. Chan, Precise Time-Synchronization in the Data-Plane using Programmable Switching ASICs, in: ACM Symposium on SDN Research (SOSR), 2019, p. 8–20.

[411] R. Kundel, F. Siegmund, B. Koldehofe, How to Measure the Speed of Light with Programmable Data Plane Hardware?, in: P4 Workshop in Europe (EuroP4), 2019, pp. 1–2.

[412] G. Bonofiglio, V. Iovinella, G. Lospoto, G. D. Battista, Kathará: A Container-Based Framework for Implementing Network Function Virtualization and Software Defined Networks, in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2018, pp. 1–9.

[413] M. He, A. Basta, A. Blenk, N. Deric, W. Kellerer, P4NFV: An NFV Architecture with Flexible Data Plane Reconfiguration, in: International Conference on Network and Services Management (CNSM), 2018, pp. 90–98.

[414] T. Osiński, H. Tarasiuk, M. Kossakowski, R. Picard, Offloading Data Plane Functions to the Multi-Tenant Cloud Infrastructure using P4, in: P4 Workshop in Europe (EuroP4), 2019, pp. 1–6.

[415] D. Moro, G. Verticale, A. Capone, A Framework for Network Function Decomposition and Deployment, in: International Workshop on the Design of Reliable Communication Networks (DRCN), 2020, pp. 1–6.

[416] T. Osiński, H. Tarasiuk, L. Rajewski, E. Kowalczyk, DPPx: A P4-based Data Plane Programmability and Exposure framework to enhance NFV services, in: IEEE Conference on Network Softwarization (NetSoft), 2019, pp. 296–300.

[417] A. Mohammadkhan, S. Panda, S. G. Kulkarni, K. K. Ramakrishnan, L. N. Bhuyan, P4NFV: P4 Enabled NFV Systems with SmartNICs, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2019, pp. 1–7.

[418] D. Moro, M. Peuster, H. Karl, A. Capone, FOP4: Function Offloading Prototyping in Heterogeneous and Programmable Network Scenarios, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2019, pp. 1–6.

[419] D. Moro, M. Peuster, H. Karl, A. Capone, Demonstrating FOP4: A Flexible Platform to Prototype NFV Offloading Scenarios, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2019, pp. 1–2.

[420] D. R. Mafioletti, C. K. Dominicini, M. Martinello, M. R. N. Ribeiro, R. d. S. Villaça, Piaffe: A place-as-you-go in-network framework for flexible embedding of vnfs, in: IEEE International Conference on Communicaotions (ICC), 2020, pp. 1–6.

[421] X. Chen, D. Zhang, X. Wang, K. Zhu, H. Zhou, P4SC: Towards High-Performance Service Function Chain Implementation on the P4-Capable Device, in: IFIP/IEEE Symposium on Integrated Management (IM), 2019, pp. 1–9.

[422] D. Zhang, X. Chen, Q. Huang, X. Hong, C. Wu, H. Zhou, Y. Yang, H. Liu, Y. Chen, P4SC: A High Performance and Flexible Framework for Service Function Chain, IEEE ACCESS 7 (2019) 160982–160997.

[423] GitHub: P4SC, `https://github.com/P4SC/p4sc`, accessed 01-20-2021 (2021).

[424] H. Lee, J. Lee, H. Ko, S. Pack, Resource-Efficient Service Function Chaining in Programmable Data Plane, in: P4 Workshop in Europe (EuroP4), 2019.

[425] Y. Zhou, J. Bi, C. Zhang, M. Xu, J. Wu, FlexMesh: Flexibly Chaining Network Functions on Programmable Data Planes at Runtime, in: IFIP-TC6 Networking Conference (Networking), 2020, pp. 73–81.

[426] A. Stockmayer, S. Hinselmann, M. Häberle, M. Menth, Service Function Chaining Based on Segment Routing Using P4 and SR-IOV (P4-SFC), in: Workshop on Virtualization in High-Performance Cloud Computing (VHPC), 2020, pp. 297–309.

[427] GitHub: P4-SFC, `https://github.com/uni-tue-kn/p4-sfc-faas`, accessed 01-20-2021 (2021).

[428] R. Ricart-Sanchez, P. Malagon, J. M. Alcaraz-Calero, Q. Wang, Hardware-Accelerated Firewall for 5G Mobile Networks, in: IEEE International Conference on Network Protocols (ICNP), 2018, pp. 446–447.

[429] Ruben Ricart-Sanchez and Pedro Malagon and Jose M. Alcaraz-Calero and Qi Wang, NetFPGA-Based Firewall Solution for 5G Multi-Tenant Architectures, in: IEEE International Conference on Edge Computing (EDGE), 2019, pp. 132–136.

[430] J. Cao, J. Bi, Y. Zhou, C. Zhang, CoFilter: A High-Performance Switch-Assisted Stateful Packet Filter, in: ACM SIGCOMM Conference Posters and Demos, 2018, p. 9–11.

[431] R. Datta, S. Choi, A. Chowdhary, Y. Park, P4Guard: Designing P4 Based Firewall, in: IEEE Military Communications Conference (MILCOM), 2018, pp. 1–6.

[432] P. Vörös, A. Kiss, Security Middleware Programming Using P4, in: International Conference on Human Aspects of Information Security, Privacy, and Trust (HAS), 2016, pp. 277–287.

[433] E. O. Zaballa, D. Franco, Z. Zhou, M. S. Berger, P4Knocking: Offloading host-based firewall functionalities to the network, in: Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), 2020, pp. 7–12.

[434] A. Almaini, A. Al-Dubai, I. Romdhani, M. Schramm, Delegation of Authentication to the Data Plane in Software-Defined Networks, in: IEEE International Conferences on Smart Computing, Networking and Services (SmartCNS), 2019, pp. 58–65.

[435] G. Grigoryan, Y. Liu, LAMP: Prompt Layer 7 Attack Mitigation with Programmable Data Planes, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2018, pp. 1–4.

[436] A. Febro, H. Xiao, J. Spring, Telephony Denial of Service Defense at Data Plane (TDoSD@DP), in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2018, pp. 1–6.

[437] A. Febro, H. Xiao, J. Spring, Distributed SIP DDoS Defense with P4, in: IEEE Wireless Communications and Networking Conference (WCNC), 2019, pp. 1–8.

[438] M. Kuka, K. Vojanec, J. Kučera, P. Benáček, Accelerated DDoS Attacks Mitigation using Programmable Data Plane, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2019, pp. 1–3.

[439] F. Paolucci, F. Cugini, P. Castoldi, P4-based Multi-Layer Traffic Engineering Encompassing Cyber Security, in: Optical Fiber Communication Conference (OFC), 2018, pp. 1–3.

[440] F. Paolucci, F. Civerchia, A. Sgambelluri, A. Giorgetti, F. Cugini, P. Castoldi, An efficient pipeline processing scheme for programming Protocol-independent Packet Processors, IEEE/OSA Journal of Optical Communications and Networking 11 (2019) 88–95.

[441] Y. Mi, A. Wang, ML-Pushback: Machine Learning Based Pushback Defense Against DDoS, in: ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2019, p. 80–81.

[442] Y. Afek, A. Bremler-Barr, L. Shafir, Network Anti-Spoofing with SDN Data Plane, in: IEEE International Conference on Computer Communications (INFOCOM), 2017, pp. 1–9.

[443] A. C. Lapolli, J. A. Marques, L. P. Gaspary, Offloading Real-time DDoS Attack Detection to Programmable Data Planes, in: IFIP/IEEE Symposium on Integrated Management (IM), 2019, pp. 19–27.

[444] GitHub: ddosd-p4, `https://github.com/aclapolli/ddosd-p4`, accessed 01-20-2021 (2021).

[445] Y.-Z. Cai, C.-H. Lai, Y.-T. Wang, M.-H. Tsai, Improving Scanner Data Collection in P4-based SDN, in: Asia-Pacific Network Operations and Management Symposium (APNOMS), 2020, pp. 126–131.

[446] T.-Y. Lin, J.-P. Wu, P.-H. Hung, C.-H. Shao, Y.-T. Wang, Y.-Z. Cai, M.-H. Tsai, Mitigating SYN flooding Attack and ARP Spoofing in SDN Data Plane, in: Asia-Pacific Network Operations and Management Symposium (APNOMS), 2020, pp. 114–119.

[447] F. Musumeci, V. Ionata, F. Paolucci, M. Cugini, Filippo Tornatore, Machine-learning-assisted DDoS attack detection with P4 language, in: IEEE International Conference on Communicaotions (ICC), 2020, pp. 1–6.

[448] X. Z. Khooi, L. Csikor, D. M. Divakaran, M. S. Kang, DIDA: Distributed In-Network Defense Architecture Against Amplified Reflection DDoS Attacks, in: IEEE Conference on Network Softwarization (NetSoft), 2020, pp. 277–281.

[449] M. Dimolianis, A. Pavlidis, V. Maglaris, A Multi-Feature DDoS Detection Schema on P4 Network Hardware, in: Workshop on Flexible Network Data Plane Processing (NETPROC@ICIN), 2020, pp. 1–6.

[450] D. Scholz, S. Gallenmüller, H. Stubbe, G. Carle, SYN Flood Defense in Programmable Data Planes, in: P4 Workshop in Europe (EuroP4), 2020, p. 13–20.

[451] GitHub: syn-proxy, `https://github.com/syn-proxy`, accessed 01-20-2021 (2021).

[452] K. Friday, E. Kfoury, E. Bou-Harb, J. Crichigno, Towards a Unified In-Network DDoS Detection and Mitigation Strategy, in: IEEE Conference on Network Softwarization (NetSoft), 2020, pp. 218–226.

[453] R. Meier, P. Tsankov, V. Lenders, L. Vanbever, M. Vechev, NetHide: Secure and Practical Network Topology Obfuscation, in: USENIX Security Symposium, 2018, pp. 693–709.

[454] Benjamin Lewis and Matthew Broadbent and Nicholas Race, P4ID: P4 Enhanced Intrusion Detection, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2019, pp. 1–4.

[455] Gorby Kabasele Ndonda and Ramin Sadre, A Two-level Intrusion Detection System for Industrial Control System Networks using P4, in: International Symposium for ICS & SCADA Cyber Security Research (ICS-CSR), 2018, pp. 1–10.

[456] J. Hypolite, J. Sonchack, S. Hershkop, N. Dautenhahn, A. DeHon, J. M. Smith, DeepMatch: Practical Deep Packet Inspection in the Data Plane Using Network Processors, in: ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2020, p. 336–350.

[457] GitHub: DeepMatch, `https://github.com/jhypolite/DeepMatch`, accessed 01-20-2021 (2021).

[458] Q. Qin, K. Poularakis, K. K. Leung, L. Tassiulas, Line-Speed and Scalable Intrusion Detection at the Network Edge via Federated Learning, in: IFIP-TC6 Networking Conference (Networking), 2020, pp. 352–360.

[459] GitHub: syn-proxy, `https://github.com/vxxx03/IFIPNetworking20`, accessed 01-20-2021 (2021).

[460] J. Amado, S. Signorello, M. Correia, F. Ramos, Poster: Speeding up network intrusion detection, in: IEEE International Conference on Network Protocols (ICNP), 2020, pp. 1–2.

[461] D. Chang, W. Sun, Y. Yang, A SDN Proactive Defense Mechanism Based on IP Transformation, in: International Conference on Safety Produce Informatization (IICSPI), 2019, pp. 248–251.

[462] W. Feng, Z.-L. Zhang, C. Liu, J. Chen, Clé: Enhancing Security with Programmable Dataplane Enabled Hybrid SDN, in: ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2019, p. 76–77.

[463] P. Kuang, Y. Liu, L. He, P4DAD: Securing Duplicate Address Detection Using P4, in: IEEE International Conference on Communicaotions (ICC), 2020, pp. 1–7.

[464] X. Chen, Implementing aes encryption on programmable switches via scrambled lookup tables, in: Workshop on Secure Programmable Network Infrastructure (SPIN), 2020, p. 8–14.

[465] GitHub: Tofino AES encryption, `https://github.com/Princeton-Cabernet/p4-projects/tree/master/AES-tofino`, accessed 01-20-2021 (2021).

[466] H. Gondaliya, G. C. Sankaran, K. M. Sivalingam, Comparative Evaluation of IP Address Anti-Spoofing Mechanisms Using a P4/NetFPGA-Based Switch, in: P4 Workshop in Europe (EuroP4), 2020, p. 1–6.

[467] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, X. Luo, Programmable In-Network Security for Context-aware BYOD Policies, in: USENIX Security Symposium, 2020, pp. 595–612.

[468] GitHub: Poise, `https://github.com/qiaokang92/poise`, accessed 01-20-2021 (2021).

[469] F. Hauser, M. Schmidt, M. Häberle, M. Menth, P4-MACsec: Dynamic Topology Monitoring and Data Layer Protection With MACsec in P4-Based SDN, IEEE ACCESS 8 (2020) 58845–58858.

[470] GitHub: P4-MACsec, `https://github.com/uni-tue-kn/p4-macsec`, accessed 01-20-2021 (2021).

[471] F. Hauser, M. Häberle, M. Schmidt, M. Menth, P4-IPsec: Site-to-Site and Host-to-Site VPN With IPsec in P4-Based SDN, IEEE ACCESS 8 (2020) 139567–139586.

[472] GitHub: P4-IPsec, `https://github.com/uni-tue-kn/p4-ipsec`, accessed 01-20-2021 (2021).

[473] T. Datta, N. Feamster, J. Rexford, L. Wang, SPINE: Surveillance Protection in the Network Elements, in: USENIX Workshop on Free and Open Communications on the Internet (FOCI), 2019, pp. 1–7.

[474] GitHub: SPINE, `https://github.com/SPINE-P4/spine-code`, accessed 01-20-2021 (2021).

[475] Y. Qin, W. Quan, F. Song, L. Zhang, G. Liu, M. Liu, C. Yu, Flexible Encryption for Reliable Transmission Based on the P4 Programmable Platform, in: Information Communication Technologies Conference (ICTC), 2020, pp. 147–152.

[476] G. Liu, W. Quan, N. Cheng, N. Lu, H. Zhang, X. Shen, P4NIS: Improving network immunity against eavesdropping with programmable data planes, in: IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2020, pp. 91–96.

[477] GitHub: P4NIS, `https://github.com/KB00100100/P4NIS`, accessed 01-20-2021 (2021).

[478] M. Liu, D. Gao, G. Liu, J. He, L. Jin, C. Zhou, F. Yang, Learning based adaptive network immune mechanism to defense eavesdropping attacks, IEEE ACCESS 7 (2019) 182814–182826.

[479] J. Deng, H. Hu, H. Li, Z. Pan, K. Wang, G. Ahn, J. Bi, Y. Park, VNGuard: An NFV/SDN Combination Framework for Provisioning and Managing Virtual Firewalls, in: IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN), 2015, pp. 107–114.

[480] H. Zhang, W. Quan, H.-c. Chao, C. Qiao, Smart identifier network: A collaborative architecture for the future internet, Networks Magazine 30 (3) (2016) 46–51.

[481] R. Kumar, V. Babu, D. Nicol, Network Coding for Critical Infrastructure Networks, in: IEEE International Conference on Network Protocols (ICNP), 2018, pp. 436–437.

[482] GitHub: AquaFlow, `https://github.com/gopchandani/AquaFlow`, accessed 01-20-2021 (2021).

[483] D. Goncalves, S. Signorello, F. M. V. Ramos, M. Medard, Random Linear Network Coding on Programmable Switches, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2019, pp. 1–6.

[484] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, K. Rothermel, P4CEP: Towards In-Network Complex Event Processing, in: Morning Workshop on In-Network Computing, 2018, p. 33–38.

[485] A. Sapio, I. Abdelaziz, M. Canini, P. Kalnis, DAIET: A System for Data Aggregation Inside the Network, in: ACM Symposium on Cloud Computing (SoCC), 2017, p. 1.

[486] G. C. Sankaran, K. M. Sivalingam, Design and Analysis of Fast IP Address-Lookup Schemes based on Cooperation among Routers, in: International Conference on COMmunication Systems and NETworks (COMSNETS), 2020, pp. 330–339.

[487] Y. Zhang, B. Han, Z.-L. Zhang, V. Gopalakrishnan, Network-Assisted Raft Consensus Algorithm, in: ACM SIGCOMM Conference Posters and Demos, 2017, p. 94–96.

[488] H. T. Dang, M. Canini, F. Pedone, R. Soulé, Paxos Made Switch-y, ACM SIGCOMM Computer Communications Review (CCR) 46 (2016) 18–24.

[489] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilbermanand, H. Weatherspoon, M. Canini, F. Pedone, R. Soulé, P4xos: Consensus as a Network Service, IEEE/ACM Transactions on Networking (ToN) 28 (2020) 1726–1738.

[490] GitHub: P4xos, `https://github.com/P4xos/P4xos`, accessed 01-20-2021 (2021).

[491] E. Sakic, N. Deric, E. Goshi, W. Kellerer, P4BFT: Hardware-Accelerated Byzantine-Resilient Network Control Plane, in: IEEE Global Communications Conference (GLOBECOM), 2019, pp. 1–7.

[492] E. Sakic, N. Deric, C. B. Serna, E. Goshi, W. Kellerer, P4BFT: A Demonstration of Hardware-Accelerated BFT in Fault-Tolerant Network Control Plane, in: ACM SIGCOMM Conference Posters and Demos, 2019, p. 6–8.

[493] L. Zeno, D. R. K. Ports, J. Nelson, M. Silberstein, SwiShmem: Distributed Shared State Abstractions for Programmable Switches, in: ACM Workshop on Hot Topics in Networks (HotNets), 2020, p. 160–167.

[494] S. Han, S. Jang, H. Lee, S. Pack, Switch-Centric Byzantine Fault Tolerance Mechanism in Distributed Software Defined Networks, IEEE Communications Letters 24 (2020) 2236–2239.

[495] GitHub: SC-BFT, `https://github.com/MNC-KOR/SC-BFT`, accessed 01-20-2021 (2021).

[496] G. Sviridov, M. Bonola, A. Tulumello, P. Giaccone, A. Bianco, G. Bianchi, LODGE: LOcal Decisions on Global statEs in Progrananaable Data Planes, in: IEEE Conference on Network Softwarization (NetSoft), 2018, pp. 257–261.

[497] G. Sviridov, M. Bonola, A. Tulumello, P. Giaccone, A. Bianco, G. Bianchi, LOcAl DEcisions on Replicated States (LOADER) in programmable dataplanes: Programming abstraction and experimental evaluation, Computer Networks 181 (2020) 107637.

[498] GitHub: LOADER, `https://github.com/german-sv/loader`, accessed 01-20-2021 (2021).

[499] H. Takruri, I. Kettaneh, A. Alquraan, S. Al-Kiswany, FLAIR: Accelerating Reads with Consistency-Aware Network Routing, in: USENIX Symposium on Networked Systems Design & Implementation (NSDI), 2020, pp. 723–737.

[500] S. Luo, H. Yu, L. Vanbever, Swing State: Consistent Updates for Stateful and Programmable Data Planes, in: ACM Symposium on SDN Research (SOSR), 2017, p. 115–121.

[501] J. Xing, A. Chen, T. E. Ng, Secure State Migration in the Data Plane, in: Workshop on Secure Programmable Network Infrastructure (SPIN), 2020, p. 28–34.

[502] GitHub: P4Sync, `https://github.com/jiarong0907/P4Sync`, accessed 01-20-2021 (2021).

[503] Y. Xue, Z. Zhu, Hybrid Flow Table Installation: Optimizing Remote Placements of Flow Tables on Servers to Enhance PDP Switches for In-Network Computing, IEEE Transactions on Network and Service Management (TNSM) (2020) 429–440.

[504] C. Kuzniar, M. Neves, I. Haque, POSTER: Accelerating Encrypted Data Stores Using Programmable Switches, in: IEEE International Conference on Network Protocols (ICNP), 2020, pp. 1–2.

[505] G. C. Sankaran, K. M. Sivalingam, Collaborative Packet Header Parsing in NetFPGA-Based High Speed Switches, IEEE Networking Letters 2 (2020) 124–127.

[506] J. Woodruff, M. Ramanujam, N. Zilberman, P4DNS: In-Network DNS, in: P4 Workshop in Europe (EuroP4), 2019, pp. 1–6.

[507] GitHub: P4DNS, `https://github.com/cucl-srg/P4DNS`, accessed 01-20-2021 (2021).

[508] R. Kundel, L. Nobach, J. Blendin, H.-J. Kolbe, G. Schyguda, V. Gurevich, B. Koldehofe, R. Steinmetz, P4-BNG: Central Office Network Functions on Programmable Packet Pipelines, in: International Conference on Network and Services Management (CNSM), 2019, pp. 1–9.

[509] GitHub: p4se, `https://github.com/opencord/p4se`, accessed 01-20-2021 (2021).

[510] I. Martinez-Yelmo, J. Alvarez-Horcajo, M. Briso-Montiano, D. Lopez-Pajares, E. Rojas, ARP-P4: A Hybrid ARP-Path/P4Runtime Switch, in: IEEE International Conference on Network Protocols (ICNP), 2018, pp. 438–439.

[511] R. Glebke, J. Krude, I. Kunze, J. Rüth, F. Senger, K. Wehrle, Towards Executing Computer Vision Functionality on Programmable Network Devices, in: ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms, 2019, p. 15–20.

[512] J. Xie, C. Qian, D. Guo, M. Wang, S. Shi, H. Chen, Efficient Indexing Mechanism for Unstructured Data Sharing Systems in Edge Computing, in: IEEE International Conference on Computer Communications (INFOCOM), 2019, pp. 820–828.

[513] Y.-S. Lu, K. C.-J. Lin, Enabling Inference Inside Software Switches, in: Asia-Pacific Network Operations and Management Symposium (APNOMS), 2020, pp. 1–4.

[514] A. Yazdinejad, R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, P4-to-blockchain: A secure blockchain-enabled packet parser for software defined networking, Computers & Security Journal 88 (2019) 101629.

[515] T. Osiński, H. Tarasiuk, P. Chaignon, M. Kossakowski, P4rt-OVS: Programming Protocol-Independent,Runtime Extensions for Open vSwitch with P4, in: IFIP-TC6 Networking Conference (Networking), 2020, pp. 413–421.

[516] GitHub: P4rt-OVS, `https://github.com/Orange-OpenSource/p4rt-ovs`, accessed 01-20-2021 (2021).

[517] S. R. Li, R. W. Yeung, N. Cai, Linear Network Coding, IEEE Transactions on Information Theory 49 (2003) 371–381.

[518] Deutsche Telekom AG: Deutsche Telekom's Access 4.0 platform goes live, `https://www.telekom.com/en/media/media-information/archive/deutsche-telekom-s-access-4-0-platform-goes-live-615974`, accessed 05-17-2021 (2021).

[519] O-RAN Alliance, `https://www.o-ran.org/`, accessed 05-17-2021 (2021).

2.7 A Survey of Scheduling Algorithms for the Time-Aware Shaper in Time-Sensitive Networking (TSN)

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 11 May 2023, accepted 9 June 2023, date of publication 14 June 2023, date of current version 22 June 2023.
Digital Object Identifier 10.1109/ACCESS.2023.3286370

SURVEY

A Survey of Scheduling Algorithms for the Time-Aware Shaper in Time-Sensitive Networking (TSN)

THOMAS STÜBER, LUKAS OSSWALD, STEFFEN LINDNER, AND MICHAEL MENTH, (Senior Member, IEEE)
Chair of Communication Networks, University of Tübingen, 72074 Tübingen, Germany
Corresponding author: Thomas Stüber (thomas.stueber@uni-tuebingen.de)

This work was supported by the German Federal Ministry of Education and Research (BMBF) through the Collaborative Project Künstliche Intelligenz zur dynamischen Optimierung des Netzwerkmanagements (KITOS) under Grant 16KIS1161. We additionally acknowledge the support from the Open Access Publication Fund of the University of Tübingen.

ABSTRACT Time-Sensitive Networking (TSN) is an enhancement of Ethernet which provides various mechanisms for real-time communication. Time-triggered (TT) traffic represents periodic data streams with strict real-time requirements. Amongst others, TSN supports scheduled transmission of TT streams, i.e., the transmission of their frames by end stations is coordinated in such a way that none or very little queuing delay occurs in intermediate nodes. TSN supports multiple priority queues per egress port. The TAS uses so-called gates to explicitly allow and block these queues for transmission on a short periodic timescale. The TAS is utilized to protect scheduled traffic from other traffic to minimize its queuing delay. In this work, we consider scheduling in TSN which comprises the computation of periodic transmission instants at end stations and the periodic opening and closing of queue gates. In this paper, we first give a brief overview of TSN features and standards. We state the TSN scheduling problem and explain common extensions which also include optimization problems. We review scheduling and optimization methods that have been used in this context. Then, the contribution of currently available research work is surveyed. We extract and compile optimization objectives, solved problem instances, and evaluation results. Research domains are identified, and specific contributions are analyzed. Finally, we discuss potential research directions and open problems.

INDEX TERMS Time-sensitive networking (TSN), time-aware shaper (TAS), scheduling, optimization, ethernet bridging.

I. INTRODUCTION
Modern applications, e.g., Industry 4.0 factory automation and motion control, demand highly deterministic network service. Exceeding latency and jitter bounds can result in immediate degradation of manufacturing quality or endanger health of machinery and operators. Some of these applications have to exchange data streams with precise timing to keep application-specific deadlines. Time-Sensitive Networking (TSN) is an emerging technology which enhances Ethernet networks with real-time properties. In TSN, talkers send uni- or multicast streams, called streams, to traffic sinks, called listeners. The network admits streams and guarantees quality of service (QoS). Time-triggered (TT) traffic constitutes periodic data streams with real-time requirements such as bounded latency or jitter. The transmission times of TT streams at their respective talkers must be scheduled such that excessive queuing in the network is avoided and their requirements are met. Although TT traffic has high priority, it can be delayed by low-priority frames in transmission blocking links for short time. To ensure that links are not occupied by low-priority traffic when needed for TT traffic, the standard IEEE Std 802.1Qbv [6] introduces an enhancement for scheduled traffic. The Time-Aware Shaper (TAS) can be implemented with this enhancement. It defines periodic time slices during which queues may send traffic to an output port and delays

The associate editor coordinating the review of this manuscript and approving it for publication was Divanilson Rodrigo Campelo.

TABLE 1. **Surveys covering related topics to this paper.**

Paper	Date	References	Focus	# Common referenced papers with this survey
Nasrallah *et al.* [1]	2018	407	Overview of TSN, DetNet, and 5G standards	14
Minaeva *et al.* [2]	2021	126	Scheduling in periodic systems	6
Seol *et al.* [3]	2021	207	Broad overview of TSN	29
Deng *et al.* [4]	2022	128	Broad overview of all topics related to AVB and TSN	17
Gavriluţ *et al.* [5]	2023	88	History of real-time Ethernet technologies, problem statements and solving techniques for various network design problems	13
This paper	2022	139	Traffic scheduling in TSN with the TAS	-

the respective traffic. In TSN, the TAS is used to protect TT traffic from other traffic classes. Therefore, TSN requires that appropriate TAS time slices are scheduled for output queues on all switches, in addition to the transmission times of all TT streams at their talkers. This combination guarantees very short delays for TT streams in TSN.

Standardization does not yet cover methods for computing such schedules. However, the topic has been examined by many publications. These research works use different methods for schedule synthesis, evaluation, and objectives for optimization. We survey the currently available literature for TSN schedule computation. The paper focuses on publications published until March of 2023 about TSN schedule planning with the TAS. Works about stream scheduling related to other technologies than TSN or for other traffic shapers in TSN than the TAS are not covered in this survey.

A. RELATED SURVEYS

To the best of our knowledge, no other review covers scheduling algorithms for TSN as its main topic. In fact, there is no survey about scheduling for TT streams for Ethernet networks, regardless of the used standard. However, there are surveys which intersect with the content of this work. Table 1 compiles the focus and the relationship of these surveys to this paper.

Nasrallah et al. [1] survey standards for low-latency communication. Besides DetNet and 5G, they also give a tutorial on the TSN standards. They reference a small number of papers related to traffic scheduling in TSN. However, they do not elaborate on their content as scheduling is not the focus of this work. Thus, only the most seminal works about scheduling from this time are referenced.

Minaeva et al. [2] give a literature summary for scheduling time-triggered real-time systems. They highlight research works from 1968 to 2020. As opposed to this work, they not only consider scheduling of streams in networks, but all systems with periodic schedules. Seminal works for TSN scheduling algorithms in the literature are mentioned, e.g., [7] and [8]. Out of 126 references, only 6 of them intersect with this work.

Deng et al. [4] review a wide range of topics about AVB and TSN from the literature of 2007 – 2021. Besides scheduling approaches, they also give an overview of reliability and security modeling, and delay analysis in the mentioned areas. As a wider range of topics is covered, only a small part of the survey is concerned with scheduling. From the 128 discussed works, 17 works intersect with this survey.

Seol et al. [3] review TSN as a whole. The authors cover publications of the years 2014 – 2020. An overview of active research directions is given, including computing routings and schedules in TSN. Not only the literature about scheduling for the TAS is summarized, but also work concerned with other queuing mechanisms, hardware, and simulation frameworks. Therefore, only a small fraction of the literature about scheduling for TAS-based queuing in TSN is surveyed. They cover 207 research works, of which 29 are included in this work. Because of the wide range of topics covered, these works are referenced for further reading but their content is not discussed.

The recent survey of Gavriluţ et al. [5] gives an excellent introduction in the history of real-time Ethernet technologies. Additionally, they present typical problems in the design of networks for time-critical applications, e.g., scheduling, routing, worst-case delay analysis, topology synthesis, and bandwidth allocation. Seminal works for each of these problems are reported and summarized. Important results are recalled. However, the focus is much broader than the scheduling problem for the TAS. Thus, many works about scheduling were not covered.

B. CONTRIBUTION

In contrast to the mentioned surveys of Table 1, we focus on papers about scheduling algorithms and related topics which use the TAS. This survey claims the following contributions:

- We give a tutorial on TSN basics.
- We define the TSN scheduling problem for TAS and modifications to it. Additionally, we introduce common solution methods used in the literature
- We survey currently available TSN literature about scheduling for the TAS.
- We identify research directions, categorize the available literature, and highlight contributions to these topics.
- We compare the available algorithms and the presented evaluations to derive open research questions in this area.

C. SURVEY STRUCTURE

This paper is structured as follows. In Section II we present a brief introduction to TSN with a special focus on the TAS. Then, we formally define the scheduling problem in TSN and give a tutorial to common solutions methods from literature in Section III. Section IV gives an overview of the state-of-the-art of TSN scheduling and categorizes the presented literature. Section V compares the presented research work with regard to modelling assumptions, optimization objective, problem instances and scalability. Furthermore, we present the publication history of the surveyed literature in Section VI. We discuss issues and open research questions in Sections VII. Finally, we conclude the paper in Section VIII.

D. LIST OF FREQUENTLY USED ACRONYMS

The following acronyms are used in this paper.

ASAP	As Soon As Possible.
AVB	Audio Video Bridging.
BE	Best Effort.
CBS	Credit-Based Shaper.
CP	Constraint Programming.
CQF	Cyclic Queuing and Forwarding.
GA	Genetic Algorithm.
GCL	Gate Control List.
FIFO	First-In-First-Out.
FRER	Frame Replication and Elimination for Reliability.
gPTP	generalized Precision Time Protocol.
GRASP	Greedy Randomized Adaptive Search Procedure.
ILP	Integer Linear Programming.
OMT	Optimization Modulo Theories.
PBO	Pseudo-Boolean Optimization.
PSFP	Per-Stream Filtering and Policing.
QoS	Quality of Service.
SMT	Satisfiability Modulo Theories.
SRP	Stream Reservation Protocol.
TAS	Time-Aware Shaper.
TSN	Time-Sensitive Networking.
TT	Time-Triggered.
VLAN	Virtual LAN.

II. FOUNDATIONS OF TSN

TSN is a set of standards for deterministic data transmission with real-time requirements over Ethernet networks. In this section, we present a short tutorial about TSN. First, we present AVB based on which TSN was developed. Then, we introduce TSN with a special focus on scheduling and the TAS.

A. AUDIO VIDEO BRIDGING

Historically, multimedia equipment was interconnected with half-duplex point-to-point links for data transmission. These links were often dedicated to a single purpose, i.e., the transmission of one specific data stream. This results in a large number of links which is expensive, hard to maintain, and error prone. Switched computer networks solved these problems. The most widely adopted technology for switched local area networks today is Ethernet. However, professional audio and video applications need bounded latencies and jitter, i.e., real-time guarantees for data streams. Switching in Ethernet networks was not designed for real-time transmissions. Therefore, the Audio Video Bridging (AVB) task group of the IEEE was founded to develop a standard to meet the requirements of multimedia applications in switched Ethernet networks.

AVB is organized in standards for time synchronization, admission control, and traffic shaping.

1) TIME SYNCHRONISATION

Network devices need a common understanding of time to ensure that all end stations in a network are able to coordinate their actions. Every AVB-capable device is equipped with a clock. The standard IEEE 802.1AS [9] defines a protocol to synchronize the clocks of all devices in an AVB network. This protocol is based on the *Precise Time Protocol* (PTP) introduced in IEEE 1588 [10] and is denoted as *generalized Precise Time Protocol* (gPTP). The gPTP defines an algorithm to select a so-called *Grandmaster* among the participating nodes of the protocol. The internal clock of the Grandmaster is used as reference clock. All other devices synchronize their clocks to the clock of the Grandmaster with time information sent from the Grandmaster. Intermediate nodes adjust the received time information to compensate propagation delays, processing delays, and different clock speeds before retransmitting them. The gPTP allows sub-microsecond precision for devices with at most seven hops distance to each other. This is needed for applications running on different end stations to synchronize their actions.

2) ADMISSION CONTROL

The Stream Reservation Protocol (SRP) introduced in IEEE 802.1Qat [11] allows senders of periodic data streams, denoted as *talkers*, to reserve bandwidth in a multi-hop Ethernet network. A talker which wants to send data advertises a new data stream to its connected bridge. This advertisement contains information about bandwidth and real-time requirements, the periodicity of the stream, and the destination MAC address. The destination may be a multicast group. The bridge forwards the advertisement if the requested resources are available. Worst-case latencies are calculated at every bridge. When the request reaches the destination of a data stream, denoted as *listener*, the listener acknowledges that it is ready, and the bandwidth is reserved along the path.

3) TRAFFIC SHAPING

Traffic shaping is the generic term for techniques that distribute packet transmissions in time. The AVB working group defines the so-called *Credit-Based Shaper* (CBS) in IEEE 802.1Qav [12]. It can be leveraged to smooth out bursts such that receiving devices are not overwhelmed. This reduces buffering and congestion in the network. The CBS is a leaky bucket traffic shaper with at least two FIFO queues for two traffic classes. These classes are denoted as class A and class B. Both queues have a credit measured in bit. Dispatching and transmitting a frame from a queue is only allowed if the credit of the respective queue is non-negative. Credit increases linearly during times no frame is transmitted and decreases linearly during transmissions. Latency bounds for streams can be guaranteed by using a special configuration for the CBS defined in IEEE 802.1BA [13]. These are specific to the requirements of the AVB domain, guaranteeing 2 ms and 50 ms for class A and B traffic in networks with at most 7 hops. However, the average delay of a frame increases to up to 250 μs per hop in the worst case when the CBS is used.

B. TIME-SENSITIVE NETWORKING

Ethernet networks are used in a wide range of industrial use cases as Ethernet is cheap and easy to implement. However, use cases such as industrial automation, in-vehicle communication or avionics have hard real-time requirements and need reliability. Data streams not meeting their deadlines may not only be worthless but impose safety risks. The latency guarantees and average delays offered by AVB fail to comply with the requirements of such use cases.

Time-Sensitive Networking (TSN) is a set of standards enhancing AVB for deterministic and reliable transmission of data over switched Ethernet networks. TSN is currently developed in the IEEE 802.1 TSN task group and adds new mechanisms for scheduling, traffic shaping, path selection, stream reservation, filtering and policing, and fault-tolerance. Most of the standards are enhancements of IEEE 802.1Q [14] which defines bridges and Virtual LANs (VLANs). We give a brief tutorial on the standards and mechanisms relevant for the scope of this survey, i.e., traffic scheduling in TSN with the TAS.

1) SIMILARITIES TO AVB

Similar to AVB, every device in TSN is equipped with a clock. TSN also uses the gPTP defined in IEEE 802.1AS [9] to synchronize clocks of all network devices. The CBS and an enhancement of the SRP are also part of TSN.

2) PATH SELECTION

TSN introduces a new mechanism for path selection in IEEE 802.1Qca [15]. In contrast to traditional Ethernet networks, it is not necessary to use Spanning Tree Protocols or Shortest Path Bridging. Paths can be computed by an arbitrary algorithm and are only limited to be trees. Thus, frames can be forwarded on an arbitrary path. Forwarding information of these so-called Explicit Trees are distributed with the Intermediate System to Intermediate System (IS-IS) protocol and stored in bridges. The Explicit Tree for the forwarding of a frame is determined by the MAC address of the root bridge of the Explicit Tree and the VLAN ID in the frame's header.

3) PRIORITIES

Every egress port of a TSN bridge is equipped with up to eight egress queues. These queues are First-In-First-Out (FIFO) queues. They correspond to the eight VLAN priorities defined in IEEE 802.1Q [14]. The VLAN tag in the header of an Ethernet frame determines the egress queue in which the frame waits for transmission. Every queue is equipped with a so-called Transmission Selection Algorithm (TSA). The TSA signals whether a frame is ready for transmission to a transmission selection mechanism. A possible implementation for a TSA is the CBS which allows frame transmissions only when the credit is positive. This selection mechanism selects the next queue from which a frame is dispatched and sent. TSN uses strict priority as transmission selection, i.e., the next frame is dispatched from the highest priority queue which signals a frame is ready for transmission.

4) FRAME PREEMPTION

High-priority traffic can be delayed due to conflicts with lower-priority traffic. IEEE 802.1Qbu [16] describes a mechanism for frame preemption in TSN which reduces such delays. Traffic is divided into preemptable frames and so-called express frames. The transmission of preemptable frames is paused and finished later if an express frame is ready for transmission. Consequently, a preempted frame is divided into fragments which are reassembled by the receiving node. The minimum size of a frame fragment is defined to be 64 byte. However, every fragment of a frame except for the last one has a trailer containing a 4 byte check sequence for error detection. Therefore, a frame can only be preempted after at least 60 byte were transmitted and the last 63 byte of a frame cannot be preempted.

5) RELIABILITY AND THE FILTERING OF DUPLICATES

Bridging in classical Ethernet networks assumes that no frames are duplicated and therefore no duplicates must be filtered. However, safety critical applications may require protection against frame loss and permanent link failures. IEEE 802.1CB [17] introduces a mechanism which allows to sent multiple copies of the same frame, possibly over disjoint paths, and to eliminate duplicates. Thus, only a single copy of the same frame is forwarded or delivered to a higher layer on an end station. This mechanism is denoted as Frame Replication and Elimination for Reliability (FRER).

6) TRAFFIC SCHEDULING

Time-triggered (TT) traffic, also denoted as *scheduled traffic*, consists of periodic data streams with hard real-time requirements such as bounded latency and jitter. The properties of TT streams such as period, maximum frame size, frames per period, as well as the range of possible transmission offsets from their respective talkers, are known in advance. The transmission times of these streams at their respective talkers can be controlled and must be coordinated to ensure that all streams meet their real-time requirements. The computation

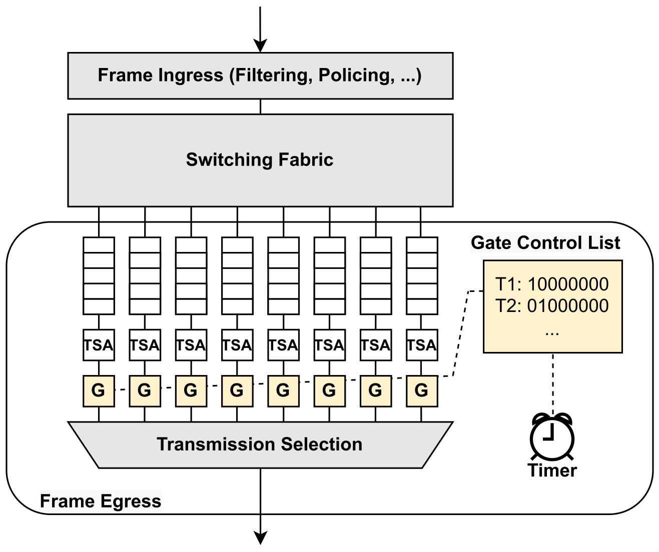


FIGURE 1. Path of a frame through a bridge. The egress port implements the enhancement for scheduled traffic. Every such egress ports has eight egress queues guarded by a transmission gate (G). The GCL controls the timed opening and closing of these gates.

of their periodic transmission times is denoted as *traffic scheduling*.

a: TRAFFIC SHAPING

TSN introduces new traffic shapers in addition to the CBS. An example for another shaper which can be used instead of the CBS is Cyclic Queuing and Forwarding (CQF) defined in IEEE 802.1Qch [18]. Time is divided into slots of predefined length. The length of a slot is denoted as cycle time. Bridges buffer all frames received during a slot and transmit them in the subsequent slot. Thus, stream latencies can easily be calculated from the cycle time and the number of hops.

IEEE 802.1Qbv [6] defines an enhancement for scheduled traffic. It can be leveraged to implement the Time-Aware Shaper (TAS). The TAS allows protecting TT traffic from other traffic such as AVB traffic or best-effort (BE) traffic. Additionally, the transit of TT streams through a network can be scheduled. Every egress queue has a so-called *transmission gate* or simply *gate*. Gates are either open or closed. Frames can only be dispatched and sent from an egress queue if the respective gate is open. The closing and opening of a gate is controlled by a so-called Gate Control List (GCL). A GCL entry consists of a time interval $[T_i, T_{i+1}]$ and a bit-vector. The bit-vector indicates which gates are opened or closed during the time interval $[T_i, T_{i+1}]$. Therefore, a GCL entries defines a time slice exclusively available to traffic with a priority corresponding to an open queue. These GCLs are executed periodically for an indefinite number of times. The computation of GCLs and appropriate cycle times, i.e., periods of these GCLs, is denoted as *scheduling* or *GCL synthesis*. The number of available GCL entries in an egress port is limited and depends on the used bridge. Figure 1 depicts the architecture of a typical TSN bridge according to IEEE 802.1Q [14], including the components of the TAS.

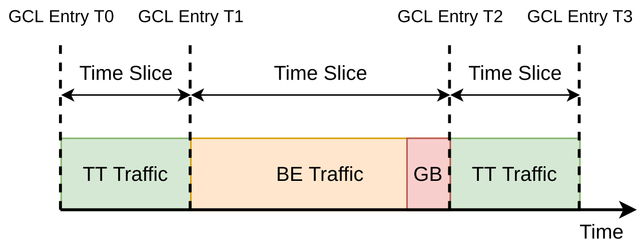


FIGURE 2. Time slices, GCL entries, and guard bands. The duration of a guard band may be not available for BE traffic as a frame can only be sent if transmission finishes before the respective gate is closed.

The TAS can be used to protect traffic by scheduling the GCLs accordingly.

b: GATE CLOSINGS AND GUARD BANDS

If the transmission of a frame is not finished until the end of the time slice the transmission started, a frame in the next time slice may be forced to wait until transmission finishes. Thus, it would be possible that a frame of a TT stream must wait because of a frame of BE traffic. This problem is avoided in TSN. Bridges detect automatically whether a frame transmission would conflict with a gate closing and hold conflicting frames back in this case. A guard band is a time interval with the length of the transmission of a maximum sized standard Ethernet frame. The duration of a guard band at the end of a time slice may not be available for transmissions to comply with closed gates. However, we emphasise that guard bands in TSN are implicit, i.e., they must not be configured explicitly. Transmissions may even start during a guard band if the transmission finishes before the next gate closing. Figure 2 depicts a guard band which restricts the transmission of BE traffic before the respective gate is closed. If frame preemption is used, the maximum size of a frame that cannot be preempted is 123 byte. This is due to the minimum size of a frame fragment, i.e., 60 byte of the frame and an additional 4 byte check sequence. A frame with 123 byte cannot be preempted until the first 60 byte are transmitted as the resulting first fragment would be too small otherwise. However, the last 63 byte also cannot be preempted as the resulting last fragment would be too small. Therefore, guard bands can be reduced to the length of a transmission of 123 byte if frame preemption is used.

c: SCHEDULER VS. TRAFFIC SCHEDULING

The term *scheduler* is sometimes used as a synonym for *traffic shaper*. For instance, the CBS and the TAS are schedulers in this terminology. Unfortunately, the term *scheduler* has also another meaning in the context of this survey. Many research works denote algorithms to plan GCL entries and frame transmissions in time with the TAS as schedulers. To avoid confusions, we will only use the second meaning in the remainder of this paper, i.e., a scheduler is a scheduling algorithm for the TAS. There are research works that use the first meaning in their title or abstract or cover scheduling algorithms for other traffic shapers in TSN, e.g., Cyclic Queuing and Forwarding (CQF). Thus, these works give the impression that they are in the scope of this survey, e.g., [19], [20], [21], [22], [23], [24], [25], [26], and [27]. However, we remark that this survey only covers research works about scheduling algorithms and the scheduling problem for the TAS. Therefore, we do not discuss works which propose new shapers, i.e., new *schedulers*, or analyse other shapers than the TAS.

The challenge in planning a TSN network is to compute the schedules that coordinate the transmission times for all streams at their respective talkers and the GCLs of bridges such that the requested real-time requirements of all TT streams are met. This problem is formally defined in the next section.

III. THE TSN SCHEDULING PROBLEM

First, we introduce a common network model, relevant properties of TT streams, and the definition of schedules. Second, we state constraints for valid schedules. Then, we discuss scheduling and optimization in the context of TSN and the computational complexity of these problems. Furthermore, we present common problem extensions solved in the literature. Finally, we give an introduction to common solution techniques that have been applied to the scheduling problem.

A. NOMENCLATURE

A node of a TSN network is an end station or a TSN-capable bridge. End stations are sources and destinations of data streams. Bridges switch frames based on their header. We remark that a node may be a bridge and an end station at the same time, i.e., it implements bridging capabilities and is an end point of data streams. Links are full-duplex Ethernet connections between an end station and a bridge or between two bridges. TSN bridges are inevitably subject to multiple delays. These delays must be considered to ensure deterministic transmissions according to a schedule. The processing delay of a bridge is the time between a frame arrives at an ingress port, and it is put in an egress queue. The transmission rate of an egress port is the rate at which data can be transmitted over a link. The propagation delay of a link is the time needed for electrical signals to traverse the link. The queuing delay of a frame is the time the frame waits in an egress queue for transmission. Ethernet uses a preamble before of a frame transmission to signal a new transmission starts, and an inter-frame gap between two frame transmissions to ensure that the receiver can process a new frame. The maximum size of a frame in TSN is 1542 byte, including inter-frame gap and preamble. A TT stream is a periodically repeated data stream with real-time requirements. Every stream has a talker as source and possibly multiple listeners as destinations. The earliest and latest transmission offsets describe the time range during which a talker can start transmission relative to the start of a period. The deadline of a stream is the time at which all frames of the stream must have arrived at all destinations, also relative to the start of a period. The entire payload of a stream must be delivered before the deadline. The payload of a stream may be sent with multiple frames.

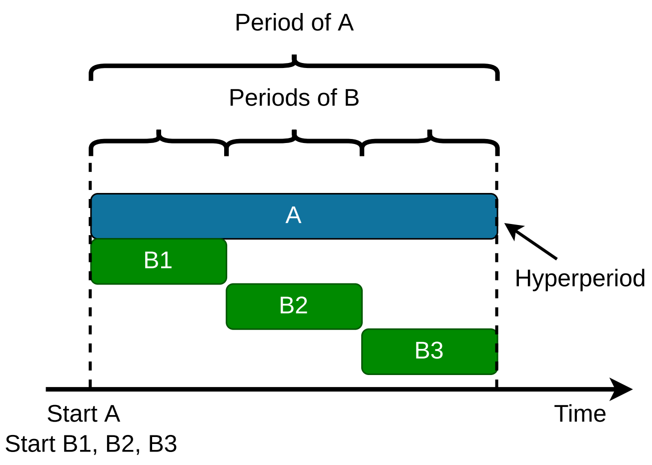


FIGURE 3. The period of stream A is three times the period of stream B. For modelling purposes, the hyperperiod is introduced, i.e., all streams are assumed to have that larger period. To cover the full duration of the hyperperiod, B is modelled by three consecutive copies B1, B2, and B3.

The hyperperiod H of a set of streams $\mathcal{S}$ is the least common multiple of the periods of the streams. Let $s \in \mathcal{S}$ be a stream with period p_s. A schedule for all streams in $\mathcal{S}$ contains $\frac{H}{p_s}$ consecutive replications of s, each having the hyperperiod as period. Scheduling algorithms typically consider transmission times, earliest and latest transmission offsets, and deadlines relative to the beginning of the hyperperiod. Figure 3 depicts an example with two streams A and B. The period of stream A is three times the period of stream B. A schedule for both streams thus contains only one period of stream A and three periods of stream B. A schedule for a set of TT streams in a TSN network consists of the transmission offsets of all streams at their respective talkers, and GCL configurations for all bridges. Transmission offsets of frames at bridges along their path follow implicitly. Schedules must be periodic, i.e., repeatable an indefinite number of times. The hyperperiod of a set of streams is the period of schedules for these streams.

B. SCHEDULING CONSTRAINTS

Given a problem instance for the TSN scheduling problem, i.e., a set of TT streams and a network topology. Every schedule which complies with the real-time requirements of all TT streams is considered a valid solution of the TSN scheduling problem. Such schedules are denoted as valid schedules in the TSN scheduling literature. The following constraints restrict the set of all possible schedules to the set of valid schedules.

1) BRIDGE DESIGN

TSN bridges are currently assumed to be store-and-forward bridges. Frames cannot be forwarded by a bridge before they have arrived at the egress queue. The duration of a transmission depends on the transmission rate of the sending egress port, the size of the frame, and the propagation delay of the used link. The processing delay of the bridge must also be considered.

2) EXCLUSIVE LINK USAGE

No two frames can be in transmission over a link in the same direction at the same time.

3) DEADLINES

A stream meets its deadline when all of its frames arrive at the stream's destination before its deadline. For multicast streams, this holds for all destinations.

4) ROUTING

The frames of a stream follow some routing. Every instance of the same stream follows the same routing.

5) FRAME ORDER

There is no reordering of frames of the same stream. Frames that are sent earlier arrive earlier than frames sent later. This holds hop-by-hop and end-to-end. The source node of a stream sends frames of a stream in-order. There are no duplicates, i.e., a frame cannot be replicated by a bridge.

6) FIFO QUEUES

The order of frame arrivals at an egress queue must match the order at which frames are sent.

7) QUEUE SIZE

Frames of scheduled traffic must not be dropped for any reason.

8) GATE CONTROL

If a frame waits in an egress queue, it can only be sent when the respective gate is open. The gate must stay open until transmission finishes.

9) TRANSMISSION SELECTION

If multiple gates of an egress port are open and frames in the respective queues are waiting for transmission, the queue with the highest priority is the next queue to dispatch a frame.

10) ADDITIONAL FEATURES

Various modifications of the problem are presented in the literature. Additional constraints may be needed to model these problems. For example, multiple queues can be reserved for TT traffic per egress port. Queue assignment of streams must be modeled in this case. We discuss these problem modifications in Section III-E.

C. FINDING A SCHEDULE VS. OPTIMIZATION

There may be multiple schedules for a given problem instance. In fact, most problem instances have a large number of schedules as possible solutions. So far, the definition of the scheduling problem does not differentiate between these solutions. A common way to compare solutions is to introduce an objective function. Such a function maps solutions to real numbers. The solution to an optimization problem is the schedule which minimizes or maximizes the objective function, i.e., has a smaller or larger objective value than any other schedule. Examples for objectives are minimizing end-to-end delays or jitter of TT streams. Another possible objective is minimizing the flowspan, i.e., the duration such that all frames have arrived at their respective destinations.

D. COMPUTATIONAL COMPLEXITY

The problem of deciding whether there is a valid schedule for a set of TT streams in a TSN network is known to be NP-complete [7] in general as Bin Packing can be reduced to it. This even holds without queuing [28]. NP is a class of decision problems, i.e., contains only problems which can be answered by either *yes* or *no*. Finding a schedule or finding an optimal schedule are not decision problems. Therefore, they are not contained in NP. However, they are computationally at least as hard as the question whether there is a schedule.

E. PROBLEM EXTENSIONS AND RESTRICTIONS

The definition of the basic problem in Section III-B only describes the common properties of the problems in the literature reviewed in this survey. Much research work focuses on special cases or problem extensions with additional constraints. This section introduces these problem variations in a general way such that they are clear in the remainder of this survey.

1) JOINT ROUTING

The definition of Section III-B assumes that the routing of every stream is a predefined part of the input and fixed. Much research work is dedicated to a variation of the scheduling problem with joint routing, which relaxes this assumption. In contrast to the basic problem, the routing of streams is variable and computed simultaneously with the schedule. This gives the scheduling algorithm more flexibility, as streams can be routed to omit heavily loaded links and thus conflicting scheduling constraints. A common approach is that the algorithm gets a set of possible paths as input for every stream, and it selects one per stream as the stream's routing. Other algorithms select arbitrary paths. Both approaches are possible due to IEEE 802.1Qca [15] as the standard allows arbitrary paths to be configured for every stream.

2) RELIABILITY

Research work dedicated to joint routing and scheduling can take reliability considerations into account. Such works define a model of possible faults and their probabilities. Scheduling algorithms can compute schedules which meet the real-time requirements of all streams with high probability for a given fault model. These schedules are denoted as robust schedules relative to a given fault model. For instance, scheduling approaches can compute schedules which are robust against single link failures. This can be achieved by introducing redundant streams with the same payload and routing them through disjoint paths.

3) GCL SYNTHESIS

GCLs for all egress ports must be constructed. One possible approach is to open the gates for scheduled traffic at the beginning of a hyperperiod and never closing them. However, this approach comes with the drawback that no other queue can send. This may be necessary to protect other TT streams with tighter bounds by avoiding congestion in the queues for TT traffic.

Another common approach is to use a postprocessing scheme after scheduling transmission offsets, e.g., in [28] and [29]. GCLs are constructed such that the gate of a queue is opened when a transmission from this queue should start according to the schedule. The respective gate is closed when the transmission is finished according to the schedule. This approach allows a scheduler to use gates to delay frames. However, the number of available GCL entries is limited in real hardware bridges. Therefore, scheduling transmission offsets and synthesizing GCLs can also be considered in a joint scheduling algorithm instead of a postprocessing, e.g., in [30] and [31].

4) QUEUING

Queuing can cause serious problems for schedules of streams with real-time requirements [29]. Frames can get lost in non-deterministic events, such as link or end station failure. A frame missing in an egress queue may result in another frame being dispatched earlier than expected and scheduled. As a result, this frame may change the arrival order in some egress queue, ultimately resulting in a stream missing its deadline. Such problems can be avoided in two ways. First, by avoiding queuing at all. Second, by not allowing frames to wait in the same egress queue at the same time. In this way, it is not possible that some frame is dispatched earlier than scheduled due to a missing frame in an egress queue. These restrictions are not imposed by bridges according to IEEE 802.1Q [14]. Instead, they are considered during scheduling such that a scheduling algorithm only computes schedules robust against these non-deterministic events. In the following, we discuss problem extensions and restrictions from the literature.

a: UNRESTRICTED QUEUING

Allowing frames of different streams to be in the same queue at the same time is denoted as unrestricted queuing. Figure 4(a) depicts a schedule by showing frame arrivals and transmissions of a single bridge. The schedule shows two streams, A and B, with two frames per period. The queuing state is shown implicitly. A frame is queued at the same time with all other frames that arrive before the frame is transmitted. Thus, the frames A1 and B1 are in the egress queue at the same time. If A1 does not arrive according to the schedule, e.g., due to a permanent link failure, B1 is transmitted earlier than scheduled. This is the case in the second period depicted in Figure 4(b). Consequently, B1 arrives earlier than scheduled in some other egress queue. This may result in some other frame experiencing more queuing delay than scheduled, ultimately leading to a missed deadline.

b: ISOLATION

The problems of queuing in case of non-deterministic events can be solved by not allowing frames of different streams to be in the same queue at the same time. If a frame is missing in a queue and no other frame is scheduled to be queued at the same time, no other frame can be transmitted earlier than scheduled. This approach is denoted as *frame isolation* in the literature. It was introduced in [29]. Figure 4(c) shows a schedule valid with frame isolation. If A1 does not arrive at the bridge, the schedule of B1 and B2 is unaffected.

c: NO-WAIT SCHEDULING

In no-wait scheduling, frames are dispatched and sent immediately after arriving at an egress queue. Queuing is not allowed. The rational of this constraint is to avoid all consequences of non-deterministic events related to queuing. It was introduced to the domain of TSN in [28]. Figure 4(d) depicts a no-wait schedule for two streams. All frames are sent immediately after arrival. For example, B1 is received and transmitted after A1 and before A2 in the same period.

d: QUEUE ASSIGNMENT

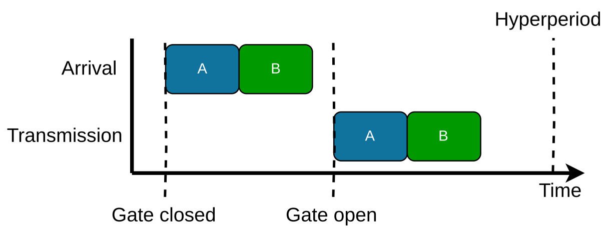
Instead of restricting queue usage, the scheduling problem can also be extended by allowing more than one queue per egress port for TT streams [29], [31]. A scheduling algorithm for such a problem not only schedules transmission offsets and GCL entries, but also assignments of TT streams to egress queues. This is especially interesting with respect to frame isolation, as frames of multiple streams can simultaneously wait for transmission by the same egress port in different queues.

5) INTEGRATION OF AUDIO VIDEO BRIDGING

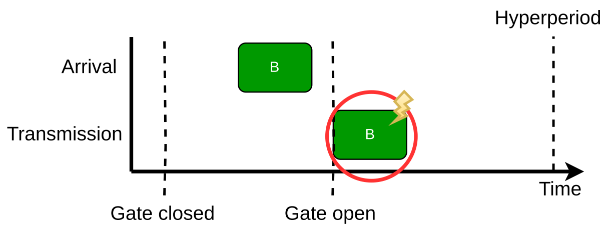
TT streams and AVB traffic can coexist in the same network at the same time. TSN bridges may support to use the CBS and the TAS in parallel according to [6]. TT streams and AVB streams compete for the same links, but use different queues in the egress ports. Therefore, the scheduling problem can be extended to also include a set of AVB streams as input. They are scheduled at their respective talkers, and considerations for the behavior of the CBS must be included during scheduling.

6) INTEGRATION OF BE TRAFFIC

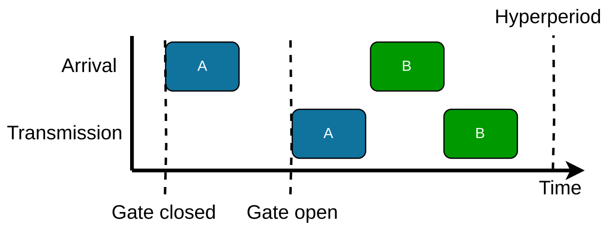
BE streams have no real-time requirements, they are generally aperiodic and unknown a priori such that they cannot be scheduled. However, some schedules may be beneficial for BE traffic. For instance, large bursts of TT traffic within a long hyperperiod could be avoided to facilitate frequent



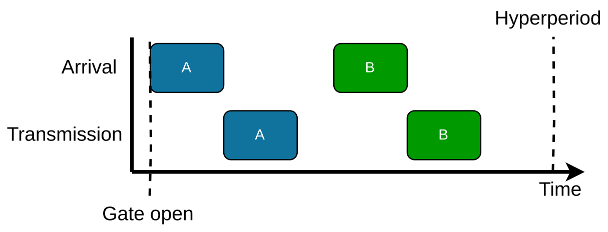
(a) Unrestricted queuing without fault. Frames A and B wait in the egress queue at the same time.



(b) Unrestricted queuing with fault. Frame B is sent earlier than scheduled if frame A was lost.



(c) Frame isolation. Frame A waits at a closed gate for some time before transmission.



(d) No-wait scheduling. Frames are transmitted immediately after arrival.

FIGURE 4. Queuing restrictions from TSN scheduling literature. Frame arrivals at an ingress port and transmissions at an egress port of the same bridge are shown. Processing delays are omitted to increase comprehensibility.

transmission opportunities for BE traffic, which may reduce the delay of BE traffic. Another example is avoiding GCL entries unless they save substantial capacity for other traffic. For each GCL entry, a guard band is needed within which frame transmissions cannot start. Therefore, compact schedules maximize capacity for BE traffic [28], [32].

7) DYNAMIC RECONFIGURATION

An entirely different problem related to the basic problem is dynamic reconfiguration of existing schedules. Such reconfigurations are necessary when streams are removed or new streams should be integrated into a schedule. While removing streams is rather easy, adding new streams to an existing schedule can be complicated for two reasons. First, the transmission offsets of already scheduled streams may have to be changed. Second, links and egress queues are occupied by earlier scheduled streams, which places constraints on possible transmission offsets for new streams. The runtime of a scheduler during reconfiguration must be very low in many scenarios, such as in automotive use cases [33]. This is due to fast changing real-time requirements and traffic patterns of safety-critical applications. Recomputing the whole schedule with offline algorithms may be computationally infeasible in such cases.

8) MULTICAST

Streams in bridged Ethernet networks can be multicast streams according to [14]. Multicast streams have more than one listener as destinations. Therefore, the routing of a multicast stream is a tree. A multicast stream can be modelled by a set of unicast streams. However, only a single copy of a frame is transmitted per hop in TSN. Thus, this modelling is not appropriate. Scheduling algorithms may contain considerations for multicast streams instead of assuming all streams to be unicast. Joint routing approaches must compute trees instead of paths for every stream.

9) TASK SCHEDULING

Tasks are applications running on end stations. They are executed periodically. Their execution depends on data received via TT streams. Additionally, they can send TT streams after they processed some received data. Scheduling algorithms for TSN can schedule tasks and TT streams in a joint approach.

F. OPTIMIZATION METHODS

We classify the scheduling algorithms in the literature in exact and heuristic approaches. Exact approaches compute a schedule, or an optimal schedule if an objective is given, if one exists, or prove the problem instance infeasible. Heuristic approaches do not guarantee to find an optimal schedule. Instead, they try to find reasonably good solutions within short time. In the common case, they cannot deduce whether a problem instance is infeasible, nor is finding a solution guaranteed if one exists. In this section, we introduce common solution techniques and explain their basics.

1) EXACT APPROACHES

As the Scheduling Problem for TSN is NP-complete, there is probably no polynomial-time algorithm to compute TSN schedules. Therefore, it is reasonable to rely on the advances of the past decades in mathematical and combinatorial optimization. All exact solution approaches in the literature are based on the following four techniques.

a: INTEGER LINEAR PROGRAMMING

An Integer Linear Program (ILP) describes the space of possible solutions to a problem with linear inequalities. Every assignment of variables which fulfills all inequalities

$$\begin{aligned} &\min 2x + 3y - z \\ \text{s.t.}\quad & 4x + 3y \leq 7 \\ & x - 2z \leq 4 \\ & 3y + 2z \leq 10 \end{aligned}$$

FIGURE 5. Example of an ILP model.

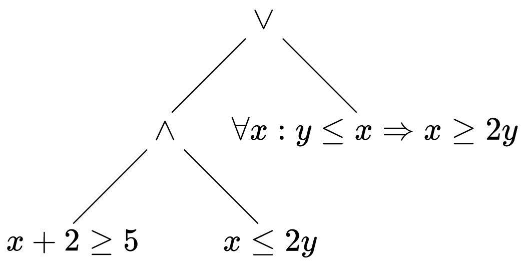


FIGURE 6. Example of a formula used in SMT solving. The basic structure is a formula from propositional logic, but predicates from other theories may be used as atomic formulas instead of Boolean variables.

corresponds to a solution of the problem and vice versa. Some variables may be restricted to take only integer values. A linear objective function may describe the quality of solutions. Figure 5 depicts an ILP which minimizes an objective function. ILP solvers compute a feasible assignment which minimizes the objective function. Every encountered solution in the solution process corresponds to an upper bound on the objective value of the optimal solution. Additionally, the solver can infer lower bounds for the objective value of the optimal solution during the solution process. So even when finding the optimal solution is not possible in reasonable time, ILP solvers yield estimations of the maximum gap to the optimum. Widely used state-of-the-art ILP solvers are CPLEX [34] and Gurobi [35].

b: SATISFIABILITY MODULO THEORIES

Satisfiability Modulo Theories (SMT) solvers find solutions to problems described by first-order formulas. Formulas model a problem with variables and predicates which are connected by logical operators. Besides Boolean variables, SMT solvers allow formulating predicates in other logical theories and use them as atomic formulas. SMT solvers have an interface for theory-specific solvers so that the problem can be modelled with the best suitable theory. Examples of theories are the theory of linear arithmetic with integers or the theory of bit vectors. Figure 6 depicts a formula with predicates from the theory of linear arithmetic with integers. The basic structure of a model is a formula from propositional logic, but predicates from integer arithmetic are used as atomic formulas.

The solver searches for an assignment of the variables that evaluate the formula to *true*. It uses techniques from SAT solving to reason about satisfiability, combined with theory-specific solvers for conjunctions of predicates. SMT solving is only about finding some satisfying solution. When the best assignment regarding some objective function is computed, the term OMT is used. Z3 [36] is a widely used SMT solver which can also be used for optimization.

c: CONSTRAINT PROGRAMMING

Constraint Programming (CP) is a general solution approach to combinatorial problems. The set of feasible solutions to a problem is described in a declarative way. In this sense, ILP and SMT solving are special cases of CP solving. However, CP solvers use backtracking, local search, and constraint propagation techniques to solve CP models as opposed to ILP solvers. Another relevant case of CP is the restriction of variable domains to a finite set. CP-SAT is a widely used CP solver [37].

d: PSEUDO-BOOLEAN OPTIMIZATION

Similar to ILPs the solution space of a problem in Pseudo-Boolean Optimization (PBO) is modelled with linear inequalities, but all variables must be binary. However, instead of using mathematical optimization as in ILP solving, techniques from SAT solving like propagation and conflict refinement are employed. A linear objective function can be minimized by adding it as a constraint to the model with some bound. The solver is called multiple times with different bounded objective constraints. Every infeasible solver run gives a lower bound on the optimal objective values. Every solution yields an upper bound on the optimal solution. The optimal solution is found, with respect to some minimal precision, when the gap between lower and upper bound is smaller than the minimal precision provided by the user.

2) HEURISTIC APPROACHES

Because finding optimal solutions for realistic problem instances is infeasible in many cases, heuristic algorithms are used. Such algorithms are used to find suitable solutions in reasonable time, generally without knowing whether there are better solutions. Metaheuristic approaches are common algorithms that can be applied for a wide range of problems. Alternatively, there are heuristics that use problem-specific knowledge for many problems, and there may be combinations of both.

a: GREEDY RANDOMIZED ADAPTIVE SEARCH PROCEDURE

Greedy Randomized Adaptive Search Procedure (GRASP) is a metaheuristic which can be adapted to various problems. Its building blocks are a greedy-randomized algorithm to construct initial feasible solutions, and a local search algorithm. The greedy randomized algorithm incrementally constructs a solution by making random decisions among the set of decisions with the smallest increase in cost until a feasible solution is found. The local search explores neighboring solutions, i.e., solutions with minimal changes, to the intermediate

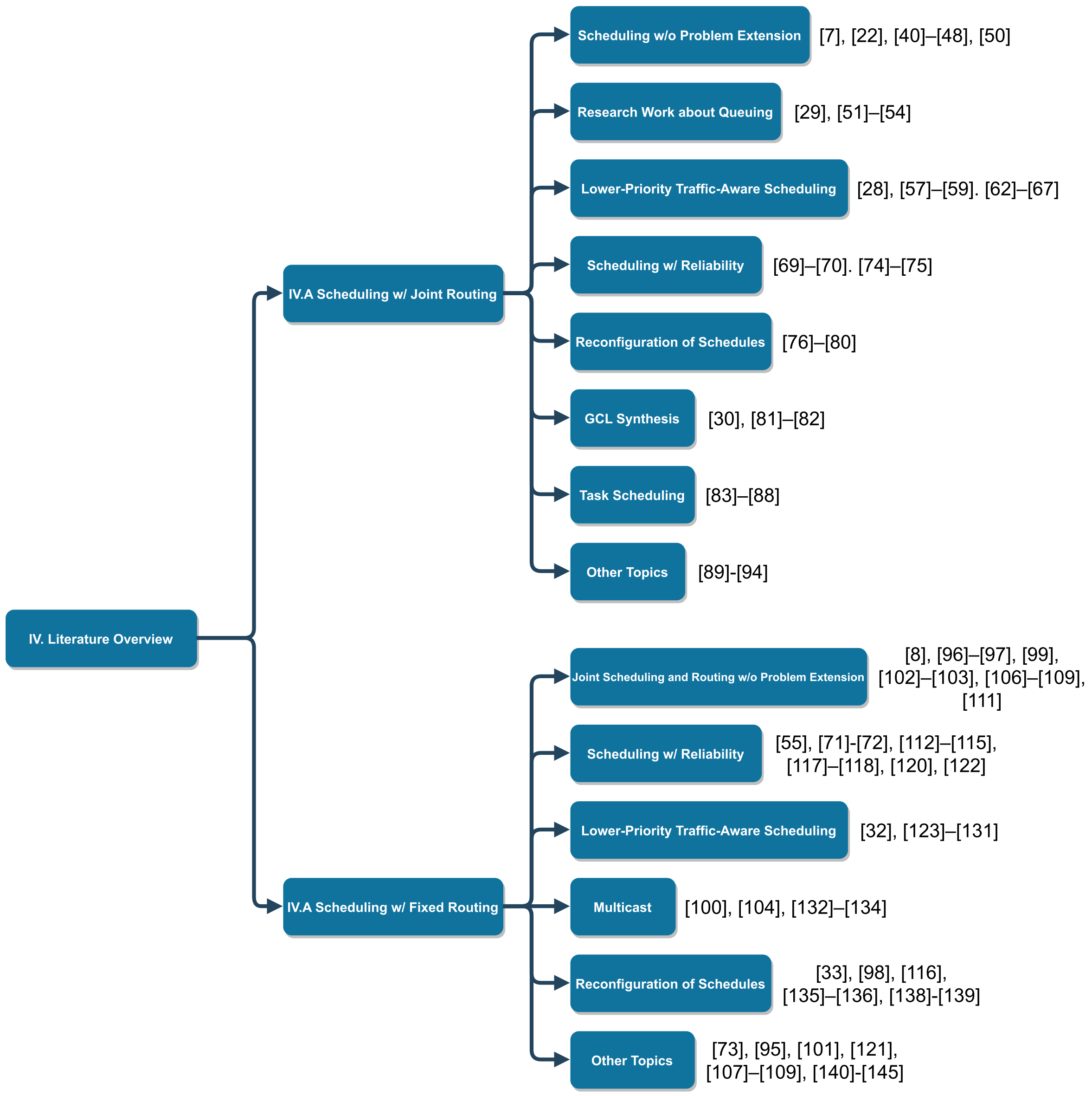


FIGURE 7. Classification of the surveyed research works in the scope of TSN scheduling.

solution. It explores the solution space until it finds a local optimum. Both steps are repeated a predefined number of times and the best encountered solution is returned. To adapt GRASP for a specific problem, a greedy randomized algorithm to generate initial solutions and a local search algorithm must be constructed.

b: TABU SEARCH

Tabu Search is a metaheuristic to systematically explore the solution space. It uses an initial solution as start and moves to the best neighboring solution. The algorithm keeps a tabu list of previously visited solutions or changes to solutions to avoid walking cycles in the solution space. Only neighboring solutions or changes to solutions which are not contained in the tabu list are considered for the move. The best encountered solution after a specific number of moves is returned. To construct a problem-specific heuristic, a heuristic to generate an initial solution and a function returning the possible changes to some given solution must be built.

c: SIMULATED ANNEALING

Simulated Annealing (SA) is a metaheuristic used to find good approximations of the global optimum of an optimization problem. It is inspired by cooling processes in physics. A global variable for temperature is used. Temperature decreases slowly to 0 in discrete steps. In each step, a neighboring solution is randomly selected, and the objective function is evaluated. The probability of moving to a neighboring solution depends on the current temperature and the objective value of the considered solution. A move to a neighboring solution which is worse than the current solution is possible with small probability to escape from local optima.

As temperature decreases, the probability of moving to solutions with worse objective value vanishes. The best solution encountered after some acceptance criterion holds is returned. To adapt SA to a specific problem, a heuristic to generate an initial solution and a function returning the possible changes to some given solution must be built. Additionally, the way the temperature is decreased and the acceptance criterion must be selected.

d: GENETIC ALGORITHMS

Genetic algorithms (GA) are a metaheuristic approach inspired by evolution processes and natural selection in biology. Candidate solutions are considered as individuals. Chromosomes represent properties of these individuals and are coded into bitstrings. At every point in time, there is a pool of individuals, i.e., the population. New individuals are constructed from two or more existing solutions, i.e., genetic crossover is performed. Individuals may be altered randomly, i.e., their chromosomes are mutated. When transiting to the next generation, some individuals die and are removed from the population. The probability of dying for an individual depends on its fitness. The fitness function is the optimization objective of the modeled problem. The best individual encountered after some number of generations is returned. As in biology, high-quality solutions have a higher probability to survive and reproduce, which in terms yields new high-quality solutions. To construct a problem-specific heuristic, a heuristic to generate initial solutions must be constructed. Suitable crossover as well as mutation and selection mechanisms have to be used. Parameters like population size, stopping criterion, and probabilities for selection and mutation must be designed.

e: LIST SCHEDULING

List scheduling (LS) is a metaheuristic to schedule tasks on identical machines. The tasks are sorted in a list according to some measure of priority. In every step, the first task in the list is selected. If a suitable machine is available, the task is executed on this machine, otherwise the next task in the list is selected. These steps are repeated until all tasks are executed. Considering streams as tasks and end stations as machines yields a heuristic for TSN scheduling. A well-known heuristic from the scheduling literature can be considered to be special case of list scheduling. As-soon-as-possible (ASAP) scheduling orders streams by priority and schedules them one by one at the earliest possible time along their paths.

f: MACHINE LEARNING

Machine learning is the generic term for a wide range of methods. Tools from linear algebra, statistics, and probability theory are used to construct mathematical models that can make decisions or construct solutions to a problem. The construction of such a model is denoted as *learning* or *training*. Typically, it takes a large amount of time and computational effort to train a model, but answers to request can be obtained really fast afterwards. Examples of machine learning methods are deep learning and reinforcement learning. However, the details of these methods are way beyond the scope of this survey. We refer to [38] and [39] for an introduction.

IV. LITERATURE SURVEY

In the following section, we give an overview of the literature about TSN scheduling. We categorize research work based on whether scheduling with fixed routing or joint routing is considered. Both sections are further grouped by the main topics of the respective papers. Comparability of techniques and results of research works in the same group is ensured by this classification. Figure 7 depicts this classification.

A. SCHEDULING W/FIXED ROUTING

We give an overview of research works which only deal with the scheduling of TT streams. In all papers presented in this section, the routing of TT streams is fixed and given as input to the scheduling algorithm. Such scheduling algorithms cannot change the routing during scheduling in case of conflicting streams. We group publications in categories based on similar topics, like model assumptions or problem extensions.

1) SCHEDULING W/O PROBLEM EXTENSIONS

We discuss publications solving the unmodified scheduling problem.

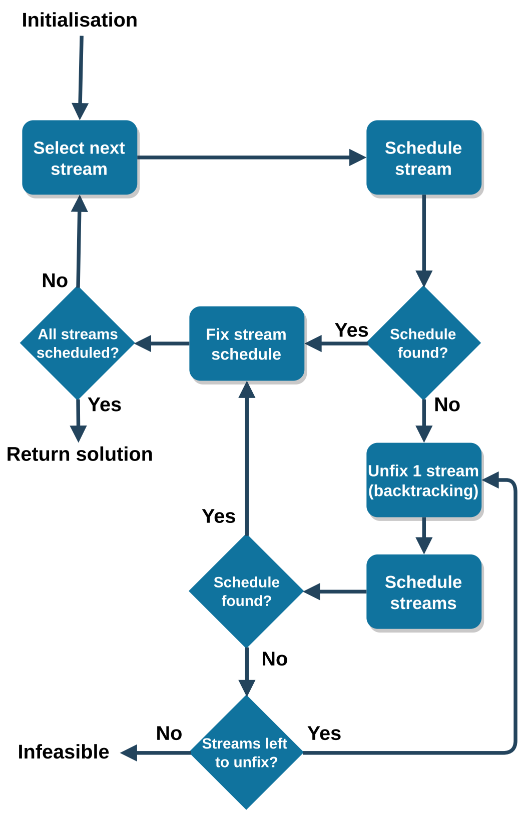


FIGURE 8. Incremental approach of Steiner [7]. Similar ideas were used by other approaches, e.g., in [29].

Early work about scheduling of TT traffic in Ethernet networks was conducted by Steiner [7]. Even though this

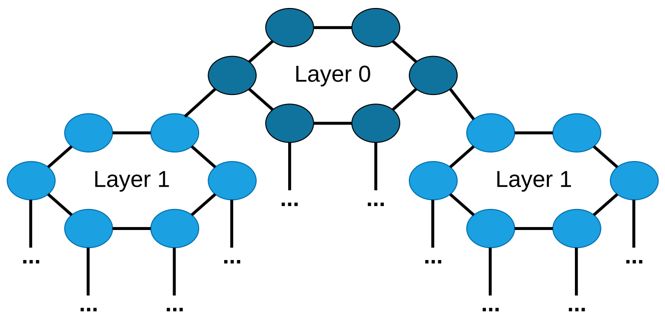


FIGURE 9. **Multi-layered ring topology used in [40].**

work is not specific to TSN, it influenced many research works covered by this survey. The author proposes the use of SMT solving for the scheduling of TT streams. An incremental approach is presented. Figure 8 depicts this approach. Streams are scheduled one after another. Schedules of already scheduled streams are fixed in later iterations. Backtracking is used in case of infeasibility, i.e., the schedule of some stream is unfixed and the stream is scheduled again simultaneously with the new stream. Backtracking is repeated until a schedule is found, or no stream schedules are left to unfix. This idea was adopted by many later works for TSN scheduling, e.g., [29] and [31].

Oliver et al. [31] give an SMT model based on mapping streams to transmission time windows of egress queues. The number of these transmission windows is fixed per egress port, and their placement and size is computed by the scheduling algorithm. As a side effect of using a fixed number of transmission windows, the number of gate events and thus guard bands is limited, even though the authors do not explore this matter. The authors use isolation to restrict the problems imposed non-deterministic behavior, e.g., frame loss. Two queues per egress port are dedicated for TT traffic. They evaluate the solving time of their approach with respect to the number of streams and the number of transmission windows per egress port. Their results indicate that the solving time increases exponentially with the number of streams. However, for reasonable numbers of streams and transmission windows, solving time is more sensible to the number of transmission windows. A comparison to the SMT from [29] shows that the window-based approach with one window per egress port is faster in finding a schedule. The average jitter is significantly reduced when the number of transmission windows per egress port is increased.

Steiner et al. [41] suggest the SMT model from [31] as a starting point for the standardization of TSN scheduling mechanisms. They demonstrate their model by reporting the same evaluation results as in [31] with a reduced number of transmission windows.

Hellmanns et al. [40] extend the Tabu Search algorithm of [28] for no-wait scheduling. They construct a 2-stage approach for hierarchical networks which consist of multiple rings on different layers. They argue that such topologies are common in factory automation. Figure 9 depicts a model of such a topology. First, they schedule streams with talker and listener in the same ring. This step is done individually for every ring. No queuing is allowed in these schedules. Then, they simulate the transmission of all streams with end points in different rings as if they were sent at the same time from their respective talkers. No queuing restrictions apply to these streams, i.e., unrestricted queuing from Section III-E4.a is allowed. If all streams meet their deadlines in this simulation, the simulated behavior is used as schedule. They compare this approach with scheduling all streams at once with the Tabu Seach algorithm. Their evaluations demonstrate that the proposed 2-stage scheduling scales better for problem instances with many streams compared to the original Tabu Search approach. The latter does not produce results for more than 1000 streams due to memory limitations. The 2-stage scheduling is two orders of magnitude faster in the special case of multi-layered ring topologies. The authors report that the number of needed GCL entries is significantly reduced by the 2-stage approach.

Another heuristic for no-wait scheduling is proposed by Zhang et al. [22]. They analyze how frame transmissions may conflict and derive the range of possible transmission offsets per frame. A comparison to the SMT of [31] and the ILP of [28] shows clear performance benefits of the proposed heuristic. The SMT was able to schedule about 300 streams, while the ILP scheduled about 1000 streams, and the heuristic scheduled 1200 streams in the evaluation scenario.

Kim et al. [42] give a heuristic algorithm to compute valid schedules, and a post-processing to reduce end-to-end delays. Streams are ordered by priority and are scheduled one after another. The individual frames of a stream are scheduled along the stream's path. The hyperperiod is divided into intervals and every frame is assigned to the earliest unoccupied interval. The presented evaluations indicate that end-to-end delays are reduced by up to one third per stream in the evaluation scenarios.

The authors of Kim et al. [43], [44] propose a genetic algorithm to schedule TT traffic in automotive scenarios. Genes encode the scheduling order of frames. Frames are scheduled as soon as possible according to this order and along the respective stream's path. The objective function used to compare scheduling orders is the weighted sum of end-to-end delays, jitter, and bandwidth utilization of the corresponding schedule. As in [28], a schedule compression algorithm is employed to reduce the bandwidth occupation of guard bands. The proposed approach outperformed random schedules regarding all three metrics in almost all evaluation scenarios. The approach from [42] is also outperformed with regard to the used objective.

Ansah et al. [45] present a scheduling algorithm in the special case of a line topology where all talkers converge in a single bridge. Based on this method, they also give an algorithm to compute GCLs in such a topology if the streams are schedulable.

The special case of an in-vehicle network with only a single hop is analyzed in [46]. They assume all traffic streams to be send continuously and belonging to different traffic classes and egress queues. Essentially, they implement round robin traffic shaping with the TAS. However, we remark that real in-vehicle networks are more complex and thus the gained insights are limited.

The authors of [47] compare the suitability of a large set of metaheuristics for TAS scheduling. They maximize the number of scheduled streams for the same problem instance with various functions of a metaheuristic library. The authors observed the best results with math based and system based heuristics and interpret this as a hint for future research directions.

Vlk et al. [48] propose a heuristic algorithm to schedule very large-scale problem instances. The algorithm shares many similarities with the well known DPLL algorithm from SAT solving [49], e.g., probing, backtracking in the case of conflicts, and restarts. Frames are scheduled one by one. If a conflict arises, all decisions are reverted up to the conflicting frame. The authors compare the heuristic with SMT-, ILP-, and GRASP-based algorithms. The schedulability of an approach with respect to some set of problem instances is the fraction of solvable problem instances within some time limit. The proposed heuristic outperforms all other approaches regarding schedulability and solving time. In fact, they were able to schedule instances with up to 10812 streams in a tree-like topology with 2000 nodes. This result outperforms all other approaches in the literature. Evaluations with a real-world instance from avionics are also presented.

Wang et al. propose a deep reinforcement learning approach for no-wait scheduling in [50]. They train machine learning models for various network topologies. The model aims to reduce the maximum arrival time among all frames to reduce the number of guard bands. For networks with up to 9 bridges and 10 end stations, the authors report solving times of at most 400 s.

2) RESEARCH WORK ABOUT QUEUING

We highlight works which allow or deal with the implications of queuing.

Craciunas et al. [29] construct an incremental SMT model to schedule TT streams based on [7]. They define flow isolation and frame isolation as properties of a schedule to prevent some sources of non-determinism, e.g., single link failures. They present models to compute schedules with either flow or frame isolation. Besides isolation in the time domain, they also employ isolation in the spatial domain by the possibility of assigning different streams to different queues. The authors identified the problem of clock synchronization errors and introduce gaps between frame transmission to cope with this problem. They compare the effect of frame and flow isolation to the solving time of their SMT model. Their evaluations indicate that flow isolation reduces solving times compared to frame isolation. However, more problem instances can be scheduled with frame isolation.

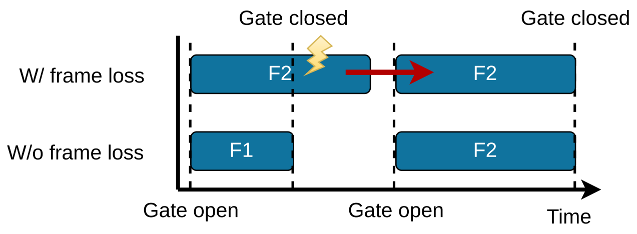


FIGURE 10. Size based isolation proposed in [53]. Frame transmissions from an egress port connected to an end station are shown. Assume F1 and F2 are scheduled to wait some time in the same egress queue. If the first frame F1 has not arrived at the egress queue, F2 cannot be transmitted in the time slice dedicated for F1 as it is too short.

Vlk et al. [51] investigate the effect of the isolation constraints from [29] on schedulability. When a frame is lost during transmission and does not reach the next egress queue as scheduled, another frame may be dispatched earlier than scheduled from this queue. This frame in term can cause more non-determinism on its path. Not allowing frames of different streams to be in the same queue at the same time solves this problem, but reduces the solution space considerably. A modification for bridges implementing the TAS is proposed to cope with this conflict. Queues with this modification check whether the next frame is the correct one with respect to the schedule. If this is not the case, the queue idles until the next frame transmission is scheduled. A comparison shows clear benefits regarding schedulability. The number of streams which are scheduled to arrive before their deadline is also significantly increased compared to isolation models.

The authors of [52] present a heuristic to schedule streams with queuing. The heuristic is based on transmission windows similar to [31]. In contrast to earlier works which include queuing in their model [29], [31], they drop isolation constraints. Network calculus is employed for a worst-case end-to-end latency analysis. They minimize the occupation percentage of egress ports, i.e., the percentage of the hyperperiod which is reserved for TT traffic. In this way, long and frequent time intervals for lower-priority traffic are scheduled. Their evaluations indicate that their approach is superior regarding end-to-end delay and schedulability of streams compared to earlier works from the same authors.

Chaine et al. [53] use queuing for jitter control. They propose to schedule streams without queuing at all egress ports except for egress ports connected to end stations. Frames are buffered in these egress ports and are released such that jitter constraints are satisfied. The authors present a novel isolation approach, denoted as *size based isolation*. Frames must be buffered in increasing frame size order if they are stored in the same queue. Two GCL entries are used to close and open the corresponding gate between two frame transmissions. In this way, frames cannot be transmitted during an earlier time slice than scheduled if another frame is missing in the queue, as earlier time slices are too short. Figure 10 depicts such a scenario. The authors give an ILP model to

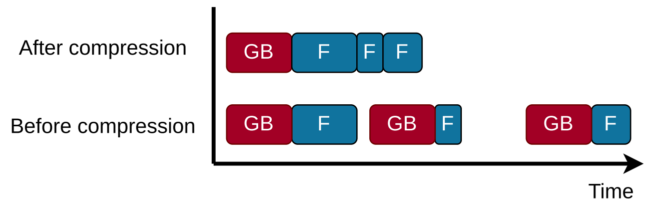


FIGURE 11. Effect of the schedule compression algorithm from [28]. Scheduled frame transmissions (F) and guard bands (GB) over a single link are shown.

compute schedules with their approach. A comparison of their approach to an unspecified approach for latency minimization demonstrates that their approach reduces scheduling time significantly. However, this comes with the cost of higher latencies.

Bujosa et al. [54] propose a heuristic scheduling algorithm which handles queue assignment of streams. Instead of scheduling frames or streams one after another on their entire path, they schedule all transmissions over a single link before scheduling the transmissions over another link. They present results about the scalability and schedulability of their approach compared to a CP approach from the literature [55]. Not surprisingly, scheduling is significantly slower with a CP approach compared to a heuristic.

3) SCHEDULING W/OTHER TRAFFIC

The schedule of TT streams may affect other traffic classes. AVB and BE traffic cannot be scheduled, but QoS metrics of these classes can be influenced when they are taken into account during the scheduling of TT streams. We summarize works with such considerations.

Dürr et al. [28] present an ILP and a Tabu Search algorithm to compute no-wait schedules for TT streams. They model the problem with job-shop scheduling, a widely used modelling framework in the scheduling literature. The authors measure the solving times of their Tabu Search and conclude that the network topology and size have no influence on solving times. They minimize the flowspan to construct a large time slice for BE traffic at the end of the computed schedules. They propose a compression algorithm as post-processing for schedules which aims to reduce the number of GCL entries needed to deploy a schedule. The authors note that this increases the available bandwidth for BE traffic as the number of guard bands is reduced. Figure 11 depicts the effect of the schedule compression algorithm. They report that the number of GCL entries can be reduced by 24% on average. Parts of the content of this work are also featured in the PhD thesis of Nayak [56].

The authors of [57] give an ILP to compute schedules for TT streams and additionally present a GRASP heuristic to schedule AVB streams. They restrict queuing by enforcing frame isolation. Their heuristic computes a routing for AVB streams such that they meet their deadlines. It reduces the search space by only considering a fixed number of shortest paths for every pair of nodes as possible routings. The schedule of TT streams, computed by their ILP model, serves as input for the heuristic and cannot be changed. They compare their AVB routing with the naïve approach of always selecting the shortest path. The comparison demonstrates that more AVB streams can be scheduled with their approach. A comparison of solving times of their ILP to the SMT from [29] is conducted. They state that their proposed ILP does not scale well for industrial-size instances and further efforts to create a suitable heuristic are needed.

The authors of [58] propose an SDN-based method for traffic bandwidth allocation in safety-critical environments. While this work does not present a scheduling algorithm for the TAS, it gives a method to configure the CBS such that latency requirements of streams are met. They use a particle swarm optimization heuristic for this purpose. Evaluation results for the schedulability of stream reservation messages under varying network utilization by TT traffic are reported.

Santos et al. [59] present an extensive SMT-based modelling of the scheduling problem with openly accessible implementation. Their model contains a range of features known from previous works, e.g., transmission windows, multicast, guard bands, and bandwidth considerations for BE traffic which were not covered by a single approach in the past. The starvation of BE traffic is prevented by restricting a user-defined fraction of a hyperperiod exclusively to be used by other traffic which is related to the approach of minimizing the flowspan [28]. Additionally, unrestricted queuing is integrated which is uncommon in exact approaches so far. The authors mention the limitation of only one gate opening per queue per hyperperiod in the presented model which reduces the available bandwidth for other traffic classes. They evaluate their approach on a realistic sized network and report successful scheduling for up to 10 multicast streams. The model is also used in the well known simulation framework OMNeT++ [60]. The thesis of Santos [61] explains the model in detail.

Houtan et al. [62] compare schedules computed with various objectives for the same SMT model with respect to the QoS of BE traffic. They propose minimization and maximization of frame offsets, with the goal of increases the QoS by grouping frames together. Additionally, they also suggest two objectives which maximize the gaps between consecutive frame transmissions over a link. They integrate frame and flow isolation in their SMT model. Unfortunately, their work lacks a description which one was used in the evaluations. A comparison of the different objective functions indicates that larger gaps between frame transmissions of TT streams increase the QoS of BE streams. For instance, BE traffic may experience less starvation and average latencies are reduced. Figure 12 depicts how BE traffic may benefit from maximizing the gaps between TT frame transmissions. However, we note that the used system model features deadlines for BE traffic, and they measure the number of deadline misses, so a comparison with the other mentioned research works is

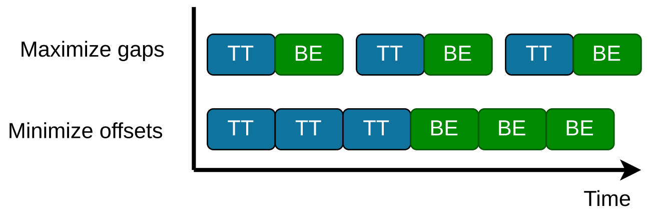


FIGURE 12. Effects of different objective functions to BE traffic in [62]. Frame transmissions of TT and BE traffic over a single link are shown.

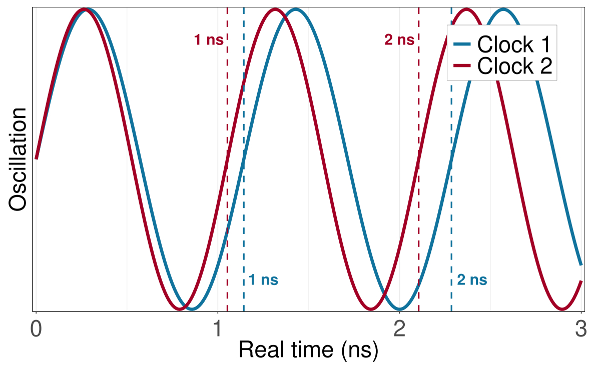


FIGURE 13. Timing signals of two clock oscillators. Clock 1 runs slower than clock 2. Therefore, the difference between both clocks increases over time.

not possible. The solving time for their SMT depends heavily on the objective function used.

The model of [29] is used for an in-vehicle scenario in [63]. The authors present simulation results with typical traffic patterns in such a scenario. They compare end-to-end delays for schedules using the TAS to schedules using the strict priority mechanism. Their results indicate that scheduling with the TAS can ensure real-time requirements of TT streams while the performance of lower-priority traffic is less affected compared to the strict priority mechanism.

Barzegaran et al. [64] give a CP approach to compute transmission windows for TT streams. In contrast to other window-based approaches [31], [41], they assume that not all end stations support TSN. They use a worst-case delay analysis to eliminate solutions that may violate the real-time requirements of the given problem instance. They compare their approach to the algorithms presented in [29], [31], [52], and [57]. They outperform these approaches in terms of solving time, but end-to-end delays and bandwidth utilization are significantly worse compared to [29] and [31]. They also perform simulation runs of their schedules with OMNeT++. The results indicate that their worst-case analysis for end-to-end delays holds but overestimates the simulated delays considerably.

The coexiestence of the TAS and Cyclic Queuing and Forwarding (CQF) in TSN is investigated by Pei et al. [65]. They propose to use CQF for rate constrained traffic with deadlines, i.e., some egress queues per egress port are shaped by CQF. Streams of scheduled traffic and rate constrained traffic are scheduled simultaneously. They are scheduled one after another in least laxity first order, i.e., the next scheduled stream is the stream which deadline expires next. The same approach is used only for scheduled traffic streams as an alternative for comparison. The evaluation shows that the joint handling of scheduled traffic and rate constrained streams results in higher schedulability.

Another approach which combines the TAS and CQF is presented by [66]. They consider the scenario of multiple traffic classes with different real-time constraints. Besides of scheduled traffic with low latency and jitter requirements, there are also two other traffic classes with uncritical periodic streams and best effort traffic. The uncritical periodic streams are assigned by egress queues shaped by the CQF. The authors present a heuristic to compute a schedule for all traffic classes simultaneously. The evaluations show that sorting streams in earliest deadline first order before scheduling is beneficial for the schedulability. In contrast to that, sorting streams by frame size or period reduces schedulability considerably.

Wang et al. [67] propose a combined scheduling scheme for TT and AVB streams. AVB streams are shaped with CQF. They use guard bands to protect TT frames from AVB frames. Their heuristic tries to schedule as many AVB streams as possible while load balancing the traffic amount between the time slots of the CQF mechanism. The authors report that their approach significantly reduces jitter and solving times compared to another approach for CQF.

Huang et al. [68] propose a recursive scheduling heuristic using backtracking. They use a complex in-vehicle topology to evaluate their approach and also include AVB streams in the evaluation scenario. We highlight that they give detailed stream parameters which is rare for real-world use cases. They also include Frame Replication and Elimination for Reliability [17] to cope with frame loss of safety critical traffic. Additionally, an SMT model is presented and compared to their heuristic. The heuristic outperforms the SMT in regard to schedulability, scalability, and end-to-end latencies by far in the evaluation scenario.

4) SCHEDULING W/RELIABILITY

Reliable transmission of data streams is one of the design goals of TSN. Additionally, to hardware features ensuring reliability, schedules can be assembled to mitigate the effects of various faults. We discuss publications which take such considerations into account.

The clock frequencies of two clocks are not exactly equal for technical reasons. This results in so-called *clock drift*, i.e., clocks running with different speeds. Figure 13 depicts this problem. Craciunas et al. [69] extend their model from [29] to cope with clock drift during scheduling. They introduce a parameter for the maximum allowed clock drift into all equations which contain transmission offsets or reception times of frames. Effectively, they merely increase the gap between frame transmissions which is already contained in the model of [29]. Clocks are resynchronized after some predefined out-of-sync detection timeout. The authors present a design space study which investigates the relationship between the maximum allowed clock drift, the worst-case clock drift rate, the maximum possible diameter of the synchronization spanning tree, and the out-of-sync detection timeout. Their findings on a number of test networks indicate that shorter out-of-sync detection timeouts are needed for higher clock drift rates. The maximum possible diameter of the synchronization spanning tree is negatively affected by higher clock drift rates. Evaluations for a test case regarding schedulability, end-to-end latency, and solving time are conducted. The results show that end-to-end latency increases for higher allowed clock drifts. Maximizing the allowed clock drift yields a maximal robust schedule for a given problem instance, but solving time increases by an order of magnitude compared to setting a fixed maximum clock drift in advance.

Feng et al. [70] consider the scheduling problem in the presence of frame loss. Instead of scheduling redundant streams over disjoint paths, as in [71], [72], and [73], reliability is achieved by multiple transmissions of a stream over the same path. Thus, the proposed approach is only applicable in the case of spontaneous frame loss and temporary link failures. The research work focuses on choosing an appropriate number of repetitions per stream for a trade-off of reliability and network utilization. In contrast to similar works, considerations for AVB and BE streams are also included in the algorithm, as repeated transmissions of TT streams deplete the available bandwidth and may lead to starvation of other traffic otherwise. The presented algorithm uses the SMT model from [29] as a sub-routine. Their results show that increasing the fault probability leads to a higher number of retransmissions which in turn results in less available bandwidth for BE traffic.

In later works, Feng et al. [74] studied a similar problem, but also considered ACK and NACK messages and queue assignment of streams. In contrast to [70], every TT stream is sent exactly twice. Transmission windows for BE streams are computed after the scheduling of TT streams. The scheduled transmission intervals for the retransmissions can be used to transmit BE traffic when no retransmissions are needed.

Dobrin et al. [75] present a heuristic scheme to schedule streams with reliability considerations. They consider transmission losses for frames such that only one frame is affected by a fault at a time and the fault is fixed by some predefined number of retransmissions. Their approach first tightens the deadlines to take some number of retransmissions into account. Then, they schedule streams in earliest deadline first order. Additional considerations for rate constrained traffic are also included in their scheme, following the scheduling of the TT streams. Unfortunately, no evaluations are presented. The authors note that future works will address more realistic fault models.

5) RECONFIGURATION OF SCHEDULES

The reconfiguration of schedules has two distinct meanings in the context of scheduling for the TAS. First, an update to an existing schedule must be computed when the set of streams or the requirements change. Second, a modified schedule from the first case must be deployed to hardware devices, i.e., GCLs and transmission offsets are reconfigured. This section focuses on the first meaning as the deployment of schedules is out of scope for scheduling algorithms. Adding and removing streams from an existing schedule is necessary in dynamically changing environments, e.g., automotive use cases. While removing a stream is straightforward, adding new streams may require more effort. We summarize research works concerned with this problem extension.

Raagaard et al. [76] propose an algorithm for online scheduling of new TT streams in an existing schedule. They use a heuristic which schedules streams as early as possible such that schedules comply with isolation. When a new stream should be added to an existing schedule, they calculate whether there is a starting offset such that the stream can be scheduled without changing the existing schedule. If this is not possible, the stream is assigned to unused queues of the egress ports along the stream's path. The authors state that adding streams to an empty schedule resembles the worst case of removing all streams from a schedule and adding a set of new streams. Thus, they evaluate how many streams can be scheduled in a specific time. They report that their heuristic is able to schedule about 1300 frames per second in medium-sized test cases.

Pang et al. [77] compute schedules with an ILP such that updating a schedule does not lead to frame loss or additional update overhead. In contrast to [28] and [57], their approach is not limited to TSN and streams are scheduled one by one. Schedules of streams from previous iterations are fixed in later iterations. When some stream cannot be scheduled, backtracking is used by removing some stream of an earlier iteration from the schedule. The authors prove that a set of additional constraints of the ILP imply no conflicts during schedule updates. They evaluate their algorithm with respect to frame loss during updates and update duration on real-world train and automotive networks. The results confirm that no frames are lost and no time overhead is needed for schedule updates.

Another algorithm for schedule updates is proposed by Wang et al. [78]. They present a heuristic scheduling algorithm with backtracking similar to [68]. Additionally, they present an algorithm for incremental schedule updates which omits frame loss during updates. A comparison between both algorithms shows that the incremental update algorithm is faster while it has poor schedulability for higher network utilization.

Gärtner et al. [79] introduce a measure for schedule flexibility denoted as *flexcurve*. The flexcurve is a function that captures the number of possible embeddings of a stream in an existing schedule. Thus, higher values correspond to more possibilities to reschedule a stream. Already scheduled streams may be selected with this measure and shifted to other times to introduce gaps for new streams into a schedule. The paper elaborates on the details of computing and updating the flexcurves. The authors compare their algorithm to a not specified SMT approach and the algorithm of Santos et al. [59]. In contrast to the SMT approach, solving time of the proposed reconfiguration algorithm is linear for up to 100 streams in the evaluation scenario. The approach of Santos et al. [59] results in schedules with lower flexibility and thus is less suitable for dynamic reconfiguration. A journal extension of this work is presented in [80].

6) GCL SYNTHESIS

Most research works use a post-processing to compute GCLs from transmission offsets. However, this comes with the drawback that GCLs have limited size in bridges and a schedule may not be deployable. We present literature which discusses explicit GCL generation.

Jin et al. [30] present an SMT approach to schedule TSN streams with a fixed number of gate openings. Reducing the number of gate events also reduces the number of guard bands, such that more bandwidth is available for lower-priority traffic. Their modelling assumptions regarding queuing are even more restrictive than frame isolation as only exactly one frame is allowed to be in a queue at any given time. Their approach allows multiple queues for TT traffic per egress port, but assigning streams to queues is not part of the SMT model. This is done before solving the SMT model by a greedy heuristic which aims to balance the workload of all queues of an egress port. As their SMT model cannot be solved in reasonable time, they use an incremental scheme to schedule small groups of streams separately. Subsets of streams are scheduled one after another such that the schedules of previously scheduled subsets are fixed in later iterations. The objective when optimizing a subset is to minimize the maximal number of GCL entries for all egress ports. They also propose a heuristic algorithm which complies with a limited number of GCL entries. Their evaluations show that the heuristic algorithm is an order of magnitude faster than naïve heuristics while reducing the number of GCL entries considerably. Instances with up to 10000 streams were scheduled in reasonable time while the SMT approach has not produced a feasible schedule for an instance with 100 streams within 2 days.

Another incremental SMT scheme which aims to reduce the number of GCL entries is given in [81]. Their approach divides the hyperperiod into slices. Streams and GCLs are scheduled for every slice individually. GCLs are updated and deployed at the beginning of every slice during schedule execution. The authors compare the number of GCL entries needed with schedules computed for an entire hyperperiod. Their results demonstrate that the number of GCL entries can be reduced while keeping end-to-end delays in reasonable bounds. However, it is not a surprise that fewer GCL entries are required when updating the GCLs regularly is allowed, as even a single entry per GCL is sufficient with frequent updates.

A rather simple CP model for scheduling on a single link with only four types of constraints is presented in [82]. However, they propose a post-processing to reduce the number of GCL entries needed in a schedule. For a small test case of only three streams, the authors report a reduction of bandwidth loss due to guard bands by 42.8%.

7) TASK SCHEDULING

Tasks are applications running on end stations. We highlight publications which consider the scheduling of tasks on end stations, additionally to scheduling data streams between these tasks.

In [83], Feng et al. compute schedules for streams and tasks sending or receiving streams simultaneously. The model includes dependencies between streams and tasks, e.g., an application can only be executed when all frames of some stream were received. The authors scheduled instances with 11 streams and more than 100 tasks. As many other works, the authors note the exponential increase of solving times for larger instances.

The authors of [84] present a CP model for scheduling of TT traffic which takes characteristics of control applications into account. Control applications have an execution interval and can only produce output streams for actuators when certain input streams of sensors have arrived. Although the quality of control application execution covers multiple aspects, the only one taken into account is jitter. Queuing is allowed in their model, but is restricted to frame isolation. They compare exact and heuristic search strategies to find solutions to the proposed CP model. For all presented test cases, both search strategies find the optimal solution with zero jitter, but the heuristic approach is orders of magnitude faster.

These preliminary works were extended in [85]. In contrast to [84], a more realistic quality measure for streams of control applications is integrated into the CP model. It constitutes of jitter and end-to-end delays of input and output streams of control applications, and jitter for control application execution. They compare their model with the model from [29] which is extended to include stream precedence for input and output streams of control applications. That means the model is able to enforce that control applications are executed after their respective input streams have arrived. Analogously, streams sent by a control application are scheduled to be transmitted after the execution of the application has finished. The authors report that the presented model outperforms the model from [29] with respect to the proposed quality measure by up to a factor of two on the test cases under consideration. Additionally, they compute a schedule for a realistic test case of an automotive mobile robot and validate their algorithm on a simulation platform and on real hardware. The PhD thesis of Barzegaran [86] features this work.

McLean et al. [87] present a converged approach for task and message scheduling in automotive environments using TSN. The authors propose metaheuristics based on genetic algorithms and simulated annealing to compute the mapping of tasks to processing cores. A combination of list scheduling

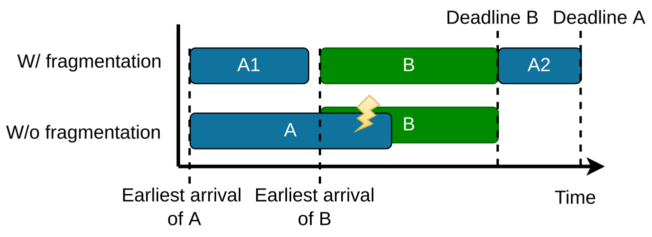


FIGURE 14. Transmissions of two frames A and B over a link to a listener. B cannot be scheduled to be transmitted at another time. In this case, scheduling is only possible when the scheduler splits A into two frames.

and earliest deadline first scheduling is used to compute schedules for message transmissions and task executions. Instead of rejecting solutions violating one or more timing constraints, the objective penalizes such solutions. Thus, the algorithm is able to move towards feasible solutions when fining an initial feasible solution is hard. The evaluations show that the simulated annealing algorithm results in lower solving times and better solutions compared to the genetic algorithm heuristic.

Another converged approach for task and message scheduling is proposed by Arestova et al. [88]. The authors introduce the concept of earliest and latest start times for tasks in cause-effect chains of tasks and streams. These times represent the range of valid execution times for tasks and are used for the fast rejection of invalid solutions. An incremental scheduling heuristic which uses these times is presented. Additionally, a repair function is integrated to recover from cases where timing constraints are violated. The evaluations demonstrate that the approach results in significantly reduced solving times and worst case response times compared to approaches from the related work.

8) OTHER TOPICS

This section summarizes research works with unique topics that fit not well into the previous groups.

Jin et al. propose an SMT model which also handles an optimized fragmentation of messages in [89]. Messages can be transmitted with multiple frames. How messages are split into frames is an additional degree of freedom in the presented optimization. Due to performance reasons when solving the model, they also give heuristics for message fragmentation and scheduling. The presented evaluations demonstrate that schedulability increases considerably by up to 50% when message fragmentation is also taken into account. Figure 14 depicts an example of how schedulability can benefit from message fragmentation. Additionally, the presented heuristic algorithms can schedule instances an order of magnitude larger than the SMT approach.

A genetic algorithm approach which takes frame preemption into account is presented in [90]. Their model contains different kinds of MAC interfaces for preemptable and non-preemptable frames. Consequently, the presented synthesis problem not only covers the assignment of streams to queues, but also the assignment of queues to interfaces. Queues are

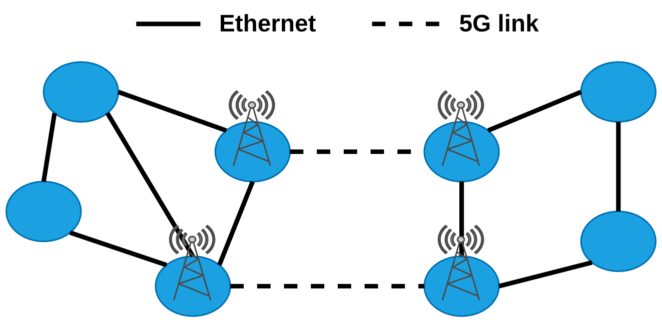


FIGURE 15. Example of a converged network with 5G and Ethernet links as considered in [91].

strictly prioritized, i.e., frames contained in a higher-priority queue always preempt frames of a lower-priority queue. The proposed GA aims to maximize the reliability of a schedule. Reliability of a stream is defined as the maximum number of allowed retransmissions without missing the deadline, and the reliability of a schedule is the minimum reliability of all streams. The authors present a comparison of the proposed GA with well-known approaches from automotive traffic scheduling. The baseline approaches are outperformed with respect to schedulability and reliability. The authors explain this result with the fact that their algorithm is specifically constructed to use all the available TSN queues and to utilize them in a way suitable for preemption.

The authors of [91] give a CP model for TSN in joint converged wired and wireless networks. Their model integrates Ethernet and 5G links simultaneously. Figure 15 depicts an example for such a converged network. Frames transmitted over a 5G link must be scheduled to fit into predefined transmission slots. They aim to minimize unusable resources in both types of links, i.e., time occupied by guard bands for Ethernet links and unused bandwidth resources for 5G links. The presented evaluations indicate that minimizing only one kind of unusable resources leads to unsatisfying results for the respective other kind.

Lin et al. [92] evaluate the impact of the so-called *network cycle* to schedulability. The network cycle is a design methodology for frame schedules. All stream periods are assumed to be integer multiples of the network cycle. Frame transmissions are aligned with network cycles. The rational of this is to omit conflicts between streams with different periods when streams are scheduled incrementally. They propose an incremental heuristic which considers the network cycle. The authors report the highest schedulability when the network cycle is set to the greatest common divisor of all stream periods.

The authors of [93] developed a graphical modelling tool for TSN scheduling. They use logic programming to deduce facts about the given problem instance. These facts are in term used for constraint generation of an SMT model. If an instance is infeasible, the conflict refinement capabilities of the SMT solver is leveraged to guide the user in changing the network configuration appropriately. Three test scenarios are presented where the streams causing infeasibility are identified.

Machine learning techniques were introduced to the domain of TSN scheduling in [94]. Tu et al. present a semi-supervised machine learning model to partition streams in groups before scheduling. They compare their approach with the partitionings in [71] and [95], and state that they are outperformed regarding schedulability. However, it is not clear how this statement is backed by the actual computation of schedules with the resulting stream groups.

We highlighted the contributions of research works for the scheduling problem with fixed routings. We compare and discuss the research works presented in this section together with the research works for the joint routing problem in Section V.

B. SCHEDULING W/JOINT ROUTING

In this section, we give an overview of research work which inspects the joint routing and scheduling problem. In contrast to works in IV-A, algorithms proposed by publications in this section compute a routing and a schedule for a given set of streams simultaneously. Again, we group the literature based on the main topic of the respective papers.

1) JOINT SCHEDULING AND ROUTING W/O PROBLEM EXTENSIONS

This section compiles publications which handle the joint routing and scheduling problem. Research works are only included when they do not focus on an additional topic highlighted in this survey.

An early ILP model which addresses the problem of joint routing and scheduling is presented in [8]. Although it is not exclusively for TSN, the authors state it is applicable for such networks. Their evaluations show that schedulability increases considerably compared to the same test cases with a fixed routing. They compare the solving time of their ILP for joint routing and scheduling with ILPs solely for scheduling. As expected, the solving time is larger for joint routing and scheduling compared to scheduling with a fixed routing. Nevertheless, they still recommend joint routing as solution quality increases considerably.

Falk et al. [96] extend the ILP from [28] to simultaneously compute routing and schedule of TT streams. They analyze the scalability of the joint routing and scheduling problem using ILPs. The authors report that solving time is more influenced by the number of streams than the size of the network topology for their ILP. The evaluations show that network topologies with more paths between any pair of nodes tend to yield harder problem instances, as more routings are possible for any stream.

Nie et al. [97] schedule and route streams incrementally. Streams are grouped by divisibility of their periods, such that streams in the same group can share the same links. The authors focus completely on the case of large period differences and omit the worst case of all streams having the same period. In contrast to similar works, e.g., [71] and [98], they consider only no-wait scheduling. Time is divided into time slots whose lengths equal the greatest common divisor of the periods of all streams. Although many evaluations are performed for different network topologies, network sizes, and traffic types, no results not seen in other works were presented.

Xu et al. [99] propose an incremental SMT scheme similar to [71] and [98]. However, they partition streams with machine learning using some of the ideas from [94]. The authors compare this partitioning approach with the partitioning algorithms from [71], [94], [95], and [100]. The best schedulability was obtained for the proposed partitioning method, second to the methods from [71] and [94]. Schedulability is slightly increased when more streams are scheduled simultaneously, as more conflicting streams are handled in the same iteration. Additionally, the authors compare the incremental scheme with their global scheduling approach from [101]. The incremental scheme outperforms the global approach with respect to schedulability and scalability. The difference in schedulability between both methods increases for higher link utilization.

The authors of [102] present a PBO model for joint routing and scheduling. They compare the solving time of their PB approach with the solving time when routing and scheduling are computed in separate steps by the same model. Their initial evaluations indicate that solving routing and scheduling in separate steps reduces the overall solving time. Surprisingly, their evaluations also show that the 2-step approach performs significantly worse for larger instances compared to the joint approach. They explain this behavior by the capability of SAT solvers to learn from conflicts. Whenever the solver runs into a conflicting variable assignment, it interferes the cause of the conflict and adds a clause to the model which prevents the conflict explicitly. The learned clauses are dropped after the routing step in the 2-step approach. Schedulability increases when routing and scheduling are performed in a single step. They state that instances with more routing options lead to easier solvable scheduling problems as streams can be distributed over the network.

Arestova et al. [103] construct a genetic algorithm for joint routing and scheduling. In contrast to other works with genetic algorithms [100], [104], the authors focus on elaborating on the construction for such an approach in detail. They combine the genetic algorithm with a neighborhood search heuristic to find better solutions efficiently. They allow queuing with flow isolation constraints from [29]. Additionally, a schedule compression algorithm similar to the one in [28] is presented, which is used to reduce the number of guard bands. In a brief evaluation section, they compare their approach with the well-known NEH algorithm [105] from job-shop scheduling. The proposed approach finds feasible schedules faster, while the resulting schedules have comparable flowspans. The authors report that scheduling with joint

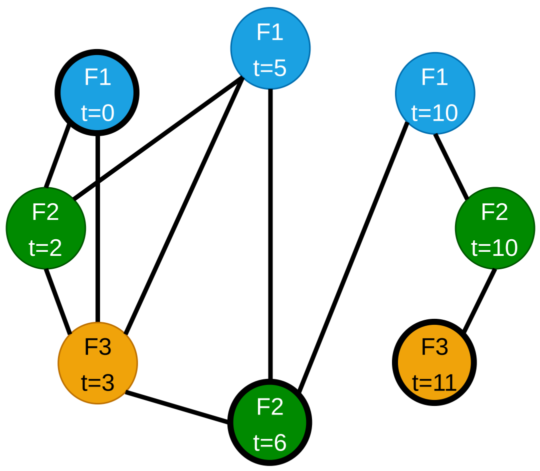


FIGURE 16. Example of a conflict graph used in [108] for a single link topology. Assume the transmission of each frame takes 5 time units. Edges indicate stream schedules which are not compatible. The black circled vertices are an independent set and induce a schedule.

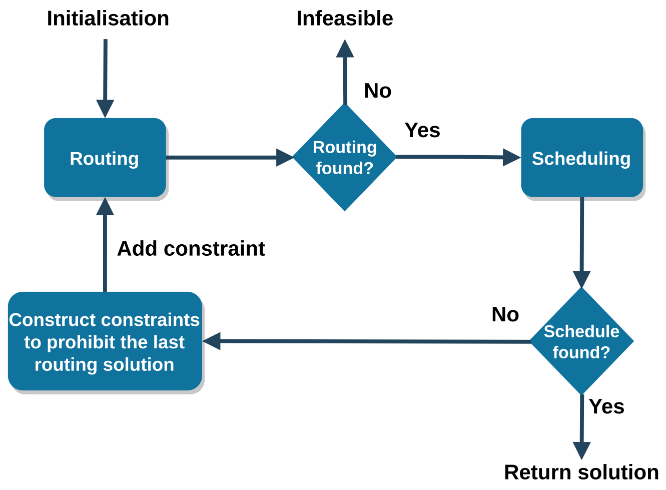


FIGURE 17. Flow diagram of the proposed approach in [109].

routing takes only slightly more time than scheduling with fixed routing with the genetic algorithm.

Kentis et al. [106] investigate the relation of port utilization and GCL schedule duration in a short paper. They employ a simple heuristic for scheduling which is not further explained, and compare the resulting schedule duration with shortest-path routing and the proposed congestion-aware routing. For most test cases, the duration of the schedule is reduced. However, they do not motivate why shorter schedule durations are beneficial.

The authors of [107] present an algorithm based on ant colony optimization. First, they give a heuristic based on simulated annealing and genetic algorithms. The authors state that integrating transmission delays into this approach is hard and propose an ant colony optimization heuristic to overcome this problem. They give the fundamental building blocks of such an algorithm and demonstrate that it can be suitable for TSN scheduling, but also note that further investigation is needed. Unfortunately, they do not elaborate on the routing, and the only related evaluation indicates that increasing the number of edges increases the solving time. The authors compare their TSN-adapted ant colony optimization algorithm with another ant colony optimization that is not specific for TSN scheduling. They conclude that the adapted algorithms results in less jitter and end-to-end delays, and converges after fewer iterations.

Falk et al. propose a joint routing and scheduling algorithm in [108] which is not based on constraints for frame transmissions. Their approach constructs a conflict graph where each vertex represents a schedule for a single stream. Vertices are connected by an edge if and only if the corresponding stream schedules are conflicting. This reduces the scheduling problem to finding an independent set in a conflict graph, i.e., a set of vertices which are pairwise not connected by an edge. Figure 16 depicts an example for a conflict graph and an independent set. The authors use incremental heuristics to construct independent sets in such graphs. Their evaluations demonstrate that the conflict graph approach has advantages regarding runtime and memory consumption compared to ILPs. They remark that their implementation is just a proof of concept, and more efficient algorithms to find independent sets are known. They further note that this approach is cheap, as no expensive ILP solver is needed.

An enhanced CP approach for routing and scheduling is presented by Vlk et al. [109]. The authors present separate models for routing and scheduling, and use them in a problem decomposition algorithm. First, they compute a routing for the given streams. A schedule is computed using this routing. If no schedule was found, constraints are added to the routing model to prohibit the last routing solution. These steps are repeated until a schedule is found. Alternatively, it may be the case that all possible routings for some stream lead to a conflict while scheduling. In that case, an instance is deemed as infeasible. Figure 17 depicts a flow diagram of the proposed approach. Most other research works which performs routing and scheduling in separate steps considers an instance infeasible after only one pass of routing and scheduling. The model also includes queue assignment of streams as additional degrees of freedom. The authors substitute their scheduling model with the algorithms from [28], [29], and [110], and compared schedulability of the resulting algorithms with their approach. Their CP model was able to schedule the most instances, while the SMT model scheduled significantly fewer instances than all other algorithms. Scheduling time was similar for all algorithms except for the SMT model, which needed multiple times longer for most instances.

He et al. [111] present a deep learning based approach for joint routing and scheduling. They use a graph neural network to handle arbitrary sized network topologies. They evaluate their approach on various random network topologies and compare it with [8], [28], [40], and [104]. All other approaches were outperformed in regard to schedulability and scalability for various numbers of streams and network topology sizes. They also compare different encodings, policies, and sampling strategies featured in their deep learning approach. The results give useful insights for future works involving deep learning in TSN scheduling. Additionally, the authors present measurements of jitter on a real hardware testbed which integrates their deep learning scheduling algorithm. They report no frame losses and ultra low jitter, which indicates that the constructed schedules are valid.

2) SCHEDULING W/RELIABILITY

Reliability is an important topic in the literature of joint routing and scheduling. The possibility of selecting disjoint paths for redundant stream copies further increases reliability in these research works.

Pozo et al. [112] present an ILP for joint routing which considers reliability constraints. After a single link failure, all streams over this link must be rescheduled and rerouted. The authors propose a fast heuristic for this case. They also evaluate which properties of a schedule are beneficial for the repairability in case of a failure. The results indicate that schedules which maximize the gaps between frame transmissions are much easier to repair than schedules which minimize the flowspan, even for three simultaneous link failures.

Atallah et al. [113] give a heuristic for fault-tolerant joint scheduling, routing, and topology generation. Their algorithm starts with a full mesh topology. Streams are routed and scheduled one after another. A k-shortest-paths algorithm is used to enumerate possible paths for a stream. If a path with available time slots is found, schedule and routing are fixed for later iterations. This is repeated multiple times with disjoint paths for redundant copies of a stream to ensure reliability. Links and bridges are only included in the final topology when they are used by some stream. The algorithm also selects bridges such that more expensive bridges are only used when necessary. Figure 18 depicts this approach for topology generation. The authors compare their algorithm to another approach which realizes redundant paths through multiple copies of the network topology. The proposed algorithm scheduled all problem instances, while it reduces financial costs for network hardware considerably.

The authors of [71] propose an incremental ILP scheme to comply with requirements for the robustness against single link failures. First, the authors present a GRASP heuristic for routing which considers reliability constraints. Streams are replicated and routed over disjoint paths to comply with requirements regarding robustness to single link failures. The resulting routing is used as input to the incremental ILP approach. Streams are partitioned into groups by introducing a conflict metric and computing a weighted cut in the conflict graph. Groups of streams are scheduled one after another. The computed schedules are fixed when the next group is scheduled. They model egress ports with only one queue for TT traffic and frame isolation. A comparison of the presented ILP with the ILPs from [8] and [102] demonstrates that it outperforms both approaches regarding solving times.

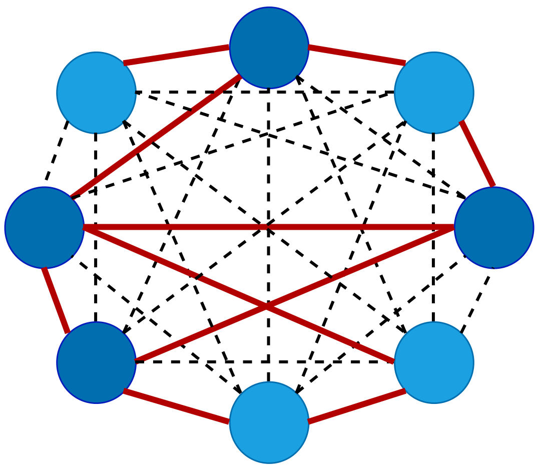


FIGURE 18. Topology generation approach of [113]. A full mesh topology is used during routing and scheduling. Links used in the final routing are indicated in red. Only these links are included in the final topology. The approach covers the selection of bridges from a library depending on the generated topology. More expensive bridges with a higher number of ports are indicated in dark blue.

Instead of robustness against link failures, Zhou et al. [114] consider reliability against frame loss, i.e., frames that are lost spontaneously during transmission without a permanent link failure. They integrate constraints regarding the loss probability along the routed path of a stream in their SMT model. However, these probabilities are only approximated, as the used theory solver cannot handle exponentials. An incremental scheme similar to [71] is used. Subsets of streams are scheduled and routed one after another until all streams are scheduled. Stream schedules are fixed in subsequent iterations. They use redundant copies of streams to further reduce the probability of frame loss, as there may be no single path with the required reliability. In contrast to other works, e.g., [71], [72], and [73], they do not enforce paths to be link disjoint. Their evaluations show that schedulability with a given level of reliability against frame loss increases with a higher number of redundant copies per stream.

Syed et al. [115] present an ILP and a heuristic for scheduling and path selection in in-vehicle networks. They leverage Frame Replication and Elimination for Reliability (FRER) [17] to ensure robustness against frame loss of safety-critical streams. The ILP model is similar to the ILP used in prior works by the same authors [116]. They report solving times of about a day with the ILP while the heuristic solved the same instances in a few minutes.

Following their works in [115], Syed et al. [117] developed an alternative to FRER. The authors propose a network coding scheme to mitigate temporary and permanent link failures. Two disjoint paths are used to transmit two frames. A third path disjoint from the other two is used to transmit the XORed data of these two frames. The loss of one frames can be tolerated as the lost frame can be reconstructed from the other two. Therefore, the redundant transmission of n frames results in $\frac{3}{2n}$ frame transmission with this scheme. This is a significant reduction compared to the $2n$ frames needed with FRER.

Another incremental scheme for scheduling and routing in safety-critical automotive applications is presented by Zhou et al. [118]. They consider possibly undetected systematic faults of bridges, i.e., implementation bugs or divergence from specifications, instead of randomly arising errors like frame loss. Messages and bridges have an Automotive Safety Integrity Level (ASIL) assigned [119] which defines reliability constraints of the respective component. A higher ASIL corresponds to higher reliability. Additionally, to computing routing and scheduling, their algorithm also selects which bridges to use from a library. Messages can be decomposed into redundant message copies with lower ASIL to be sent over bridges with lower ASIL. A comparison of the presented algorithm with and without message decomposition shows that the total financial costs for bridges can be reduced by up to 23.55% in the evaluation scenarios. The authors note that higher ASILs increase the required number of message copies, which leads to more congestion in the network and thus higher end-to-end delays. Evaluations on a real-life automotive test case are presented. Synthesis time increases considerably with ASIL decomposition. However, the selected bridges cost only a third compared to using only bridges with the highest ASIL. The PhD thesis of Zhou [120] compiles the content of [114], [118], and [121].

The authors of [72] present separate CP models to compute schedules and routings as work-in-progress. These models take security and reliability considerations into account. The routing computed with the first model is used as fixed input for the scheduling model, similar to [71]. Redundant streams with disjoint paths are added if needed to comply with security and reliability constraints.

The ideas from [72] are extended in [55]. Besides the CP model, a simulated annealing algorithm combined with a list scheduler is given. Additionally, a post-processing for latency reduction of scheduled streams is applied after scheduling. Their evaluations without security and reliability constraints indicate that the SA approach is able to find a feasible schedule in reasonable time, even for huge problem instances. However, they also introduce a measure for schedule and routing costs. This measure contains stream latencies, penalties for streams which were not scheduled, and penalties for overlapping paths. The results demonstrates that schedules computed by SA have up to three times higher costs compared to schedules computed by the CP approach. Introducing the security and reliability constraints to the same problem instances increased the costs of schedules computed by SA by up to a factor of ~ 3.5. Nevertheless, the authors state that the SA approach can be useful in comparing costs and reliability capabilities of topologies, or to reconfigure the network in case of link failures.

Li et al. [122] propose a heuristic for joint routing and scheduling with reliability constraints. A greedy algorithm is used to select paths for redundant copies of streams such that link utilization is balanced. Frames are scheduled as soon as possible. Both algorithms are combined in an iterative local search scheme. Already scheduled and routed streams are randomly removed from time to time, and the remaining streams are rescheduled and rerouted in random order. While the authors report a reduction in frame losses by their approach, end-to-end delays are significantly larger compared to schedules with routings computed by Dijkstra's algorithm.

3) SCHEDULING W/OTHER TRAFFIC

This section presents research works concerned with scheduling in the presence of other traffic classes.

Gavriluţ et al. [123] construct a GRASP heuristic to schedule TT streams while taking AVB traffic into account. In contrast to [57], their approach handles TT and AVB streams simultaneously. The routing of AVB streams is given and cannot be changed. The problem of finding a routing for AVB streams in the presence of TT streams was addressed and evaluated by Laursen et al. [124]. The authors of [124] propose a hill climbing based heuristic algorithm. In the first evaluation of [123], the authors use a shortest paths algorithm to compute the routing for TT streams. The results show that using the GRASP heuristic with an objective that considers tardiness of AVB streams leads to AVB streams meeting their deadlines. In contrast to that, the GRASP heuristic with other objectives not considering AVB streams results in schedules with late AVB streams. Even better results are obtained when a routing with load balancing is used before scheduling, which also decreases overall runtime. They report that AVB traffic does not benefit from minimizing the number of queues for TT streams. However, this is rather obvious, as their system model assumes that AVB streams already have an assignment to AVB queues. Solving time of the GRASP heuristic increases considerably when AVB streams are taken into account. A short preview of these results was previously published in [125] and the implementation details of the heuristic are elaborated in [126].

The routing algorithm in [124] only considers AVB streams in an offline scenario. Another work which focuses on the routing of AVB streams in the presence of scheduled traffic is presented in [127]. The authors propose an online routing algorithm for AVB and TSN streams. The algorithm is based on ant colony optimization and compared to the approach of [124]. The evaluations indicate that the ant colony optimization algorithm outperforms the approach of [124] with respect to solving times.

Gavriluţ et al. [128] propose an algorithm which assigns messages to traffic classes, i.e., whether a message should be transmitted with TT or AVB streams and which AVB class should be used. The assignment algorithm is based on the tabu search metaheuristic. Additionally, the parameters for the CBS are estimated such that AVB streams comply with their real-time requirements. Afterwards, streams are scheduled with an adaption of the algorithm from [123] and [126]. The evaluations show that the assignment algorithm significantly increases the number of schedulable streams. Compared to the SMT approach of [29], the assignment to traffic classes increases the schedulability of legacy streams that cannot be reassigned or rescheduled.

Berisa et al. [129] propose a heuristic for the joint routing and scheduling of TT streams in the presence of AVB streams. They make use of frame preemption to increase the schedulability of AVB streams. To that end, they present a worst-case response-time analysis of AVB streams with preemption. The heuristic is based on prior works by Gavriluţ et al. [123]. Their evaluations demonstrate that the schedulability of AVB streams can be increased by allowing frame preemption while the runtime of the heuristic also increases significantly in this case.

Alnajin et al. [32] give a QoS-aware routing algorithm for TSN streams with respect to various metrics. They present four scheduling heuristics combined with these routings. They compare these algorithms regarding the number of guard bands needed in the resulting schedules. Their evaluations show that their heuristics can reduce the number of guard bands significantly. They note that reducing the number of guard bands is beneficial for BE traffic.

Li et al. [130] present a heuristic for joint routing and scheduling to eliminate non-deterministic queuing delay in networks with mixed time-critical traffic. Their scheduling algorithm divides the bandwidth resources into time slots and assigns streams to these slots such that transmission conflicts cannot arise. Similar to [31], the maximum length of the resulting GCLs is bounded instead of being computed by a post-processing from transmission offsets. The solving time with the heuristic is compared to [71], [96], and [103]. Solving time is the only inspected metric as the presented algorithm, and the three compared approaches have no objective functions. In the presented scenarios, all three methods were outperformed by multiple orders of magnitude. The authors report schedules for 4000 streams with just 12 GCL entries per egress port on average.

Yang et al. [131] use deep reinforcement learning for the joint routing and scheduling problem. Additionally, they take AVB and BE streams into account. The authors elaborate on the details of their machine learning model and present evaluations about the learning phase. They introduce three baseline approaches also based on machine learning for comparison. The proposed model results in slightly lower average latencies for all traffic classes in the evaluation scenario.

4) MULTICAST

This section highlights research works specifically concerned with multicast streams in a joint routing and scheduling approach.

The joint routing and scheduling model from [8] is extended for multicast support in [132]. The authors state that while this extension is trivial for pure scheduling models, joint routing and scheduling with multicast is more complicated as additional constraints for the routing are needed. Various pre-processing steps are presented to reduce the solution space and thus solving time. The authors report that the time to find a feasible solution was reduced by up to 82.4% while the overall solving time was reduced by up to 47.6%.

Another approach for joint routing and scheduling with multicast streams is presented by Li et al. [133]. Similar to [132], the authors use pre-processing to simplify solving of the model. The streams are divided into groups by spectral clustering based on their properties. Similar to [71] and [72], these groups are routed and scheduled one after another such that previously computed schedules and routes are fixed. The authors report that random clustering result in slightly longer flowspans. As in similar incremental approaches, reduced overall solving times and increased schedulability are reported.

Yu et al. [134] propose an incremental approach with ILPs. In contrast to [132], they route and schedule multicast streams one by one. Multiple queues per egress port and queue assignment are also integrated in their model. Additionally, they propose a pre-processing scheme which aims to simplify the topology. The pre-processing merges cliques in the topology to a single link. If routing and schedule can be computed, both are modified for the original graph. Otherwise the conflicting links are expanded and routing and scheduling are repeated. Compared to [7] with a Steiner tree as fixed routing, the proposed approach can schedule significantly more instances.

A biology-inspired algorithm is given by Pahlevan et al. [104]. They construct a genetic algorithm for joint routing and scheduling which also comprises features like multicast streams and dependencies between streams. The authors state multicast capabilities as one of their main contributions, but consider multicast streams simply as multiple unicast streams. In contrast to [29], [31], and [125], only a single queue per egress port is dedicated to TT traffic. While their evaluations indicate that solving time increases compared to scheduling with a fixed routing, they show that schedulability increases by joint routing and scheduling.

In later works, Pahlevan et al. [100] present a heuristic list scheduling algorithm for the same purpose. They model queuing and multicast streams in the same way as in [104]. Their evaluations again demonstrate that joint routing and scheduling increases schedulability.

5) RECONFIGURATION OF SCHEDULES

Reconfiguration of streams can benefit from modifying not only a schedule, but also the respective routing. Newly added streams can be routed over paths with low utilization. This section compiles the literature about reconfiguration in joint routing and scheduling apporaches.

Research work from Syed et al. [116] deals with joint routing and scheduling in in-vehicle networks. They propose an ILP model for streams that are known in advance which takes load balancing into account. They compute schedules and routings for these streams such that as many streams as possible can later be added dynamically. The evaluations compare algorithmic details that are hard to asses without detailed knowledge of the presented approach.

Following their work in [116], Syed et al. present multiple heuristics for the dynamic scheduling of new streams in an existing schedule [135]. Their heuristics are based on modelling the scheduling problem as a vector bin packing problem. They evaluate the time needed for adding new streams in an automotive use case, as reconfiguration in such scenarios has strict timing requirements.

The authors of [33] construct multiple heuristics for dynamic scheduling and routing with reliability constraints. All heuristics are based on the same idea as in [135]. Every stream is replicated twice when added to an existing schedule to ensure reliability. The best heuristic is able to schedule 500 streams in about 410 ms. The authors state this is a reasonable response time in automotive use cases.

Yu et al. [136] developed a heuristic for online rescheduling in scenarios with virtual machines as communication endpoints. Virtual machines in a cloud computing environment may be migrated from one physical device to another such that schedules and routings must be updated. Additionally, all streams are multicast streams, which complicates rescheduling after a VM migration. Therefore, the multicast tree for a stream is computed such that the maximal distance from any possible device where a VM could run to any destination is minimized. The authors state that this will reduce conflicts when a VM is migrated, as the new paths are short. Given a schedule and a stream that is migrated, a greedy heuristic computes the new schedule based on the precomputed multicast tree. The authors compare their proposed routing heuristic with an optimal routing obtained by an ILP. Solving times are significantly reduced, while the routing objective grows only slightly compared to the optimal routing. Schedulability in case of a migration is considerably increased in comparison to the same scheduling heuristic used with a routing computed by the KMB algorithm [137].

Li et al. [138] consider the reconfiguration of routing, scheduling, and mapping of applications to end stations in case of permanent end station failure. They extend the ILP of [139] to schedule applications to end stations for global reconfiguration. As the resulting ILP instances are hard to solve, they propose a heuristic routing and mapping algorithm as alternative. The results of this heuristic are fixed in the ILP model, such that only a schedule is computed. The heuristic approach is able to reconfigure almost all instances, while the ILP times out for most of them in their evaluations. While both algorithms have exponential runtime in the number of streams, the heuristic is two orders of magnitude faster for the considered instances.

An incremental approach which schedules streams one by one is presented in [98]. In contrast to [134], the computed schedules are constrained to no-wait scheduling, i.e., no queuing delays are allowed. The authors compare the proposed approach with [8] and [71] with respect to schedulability and show that schedulability is slightly increased. The proposed pre-processing for the routing approach gives only minor improvements regarding schedulability. The authors report that 97.5% of the streams in an instance with 2000 streams were scheduled in less than 10 seconds per stream. They state this is fast enough for online scenarios.

6) OTHER TOPICS

This sections summarizes research works with unique topics that do not fit well into the previous groups.

The authors of [121] propose a heuristic model to schedule streams in the presence of frame preemption similar to [90]. Additionally, they also include route computation in their algorithm. They present an SMT model for this purpose and use it in an incremental approach, similar to [95], [102], and [140]. The presented results show that scheduling time not only increases with the network size and the number of streams, but also with the maximum number of allowed preemptions and retransmissions. However, allowing more preemptions increases schedulability only to some instance-specific threshold.

Gavriluţ et al. [73] give multiple algorithms to simultaneously compute scheduling and routing of TT streams. In contrast to [71], [72], and [96], these algorithms additionally generate the network topology with minimal financial costs imposed by network hardware. They present a problem-specific heuristic, a GRASP heuristic, and a CP approach, and compare them to each other regarding solving time and solution quality. Their optimization objective captures worst-case end-to-end delays as well as topology costs, i.e., costs for links and bridges which are selected from a library. Redundant copies of streams are included for reliability considerations. Their evaluations focus on a comparison of the three presented algorithms. As expected, the CP approach does not scale well. The GRASP heuristic finds better solutions in minutes compared to the CP approach in two days.

An SMT model which includes scheduling, routing, and queue assignment of streams simultaneously is presented by [101]. The authors state that saving bandwidth by not using the same GCL cycle for all egress ports is also novel to their approach. However, this is not true as other works even schedule GCL closing events, e.g. [30] and [41]. They propose to minimize the number of bridges used by scheduled traffic in order to maximize utilization. In comparison to the list scheduler of [100], schedulability is increased while the solving time approaches the timeout after 40 h for fairly small instances.

Zhang et al. [141] construct a heuristic which allows different routes for frames of the same stream to enable load balancing. The required mechanism is implemented by an SDN architecture. The scheduling procedure is a mix of evolutionary algorithm and greedy algorithm. Multiple variations of the heuristic are compared in the evaluations. In the presented scenarios, scheduling time increased linearly with the number of streams.

Another incremental scheme for scheduling and routing is presented by Mahfouzi et al. [95], [140]. The authors investigate the stability of control applications, i.e., latency and jitter of messages sent by these applications. Instead of grouping streams by some conflict measure, they divide the network period in slices and group the streams by the time slice in which their transmission can start. They allow routing only over some fixed number of precomputed shortest paths per source-destination pair, similar to [123] and the AVB routing in [57]. Their model allows unrestricted queuing without further discussion of this topic. The evaluations indicate that the number of allowed paths has a huge impact to solving time. More possible paths result in a significant increase in solving times. However, they note that three paths per pair of nodes may be sufficient, as schedulability is over 90% in this case. They conclude that the search space can be reduced without decreasing schedulability considerably.

Yang et al. [142] present a network architecture for industrial use cases based on TSN hardware and software-defined networking (SDN). They focus on so-called chain flows. Chain flows consist of multiple stream which are joined at one or more nodes. For instance, an industrial controller may join streams from multiple sensors and forward a single stream to an actuator. The authors propose a tabu search heuristic and an ILP for the scheduling of chain flows. They report benefits in scalability and schedulability in comparison to handling every stream of a chain flow individually.

Chain flows are further investigated by Gong et al. [143]. They propose a heuristic time-tabling algorithm combined with a tabu search for schedule reconfiguration when the network topology is changed. In contrast to the magazine article [142], they elaborate on the details of these algorithms. However, the reported results are consistent with the results in [142].

Hellmanns et al. [144] focus their work not on the ability to compute schedules, but analyze how input pre-processings and solver configuration influence the scalability of solving a joint routing and scheduling ILP model. They categorize optimizations by whether they are input pre-processing, e.g., topology reduction, model generation related, e.g., tighter variable bounds, or solver configurations, e.g. the use of value hints for variables. They give an ILP without any optimizations as baseline for their evaluations. Different combinations of the proposed optimizations are tested on the same set of problem instances and compared with respect to scalability and schedulability. Their evaluations indicate that solving time can greatly benefit from input pre-processings, but the effects of model generation optimizations and solver configurations are negligible. Some of the optimizations even increase solving time. However, queuing is not supported by their base model. Thus, the observations only hold for the no-wait case without queuing delay. It is not clear whether these results can be transferred to other problem extensions from the literature.

Bhattacharjee et al. [145] propose two algorithms for the placement of talker applications in a network. Additionally, both approaches also solve the joint routing and scheduling problem. The first algorithm is an ILP for the placement of talker applications that is combined with the GRASP heuristic of [123]. The second algorithm is a simulated annealing (SA) heuristic which computes the placement of talker applications, the schedule, and the routing for a given problem instance. The evaluations show that both algorithms behave approximately similar with respect to load balancing and solving times. However, the authors report considerably reduces stream latencies with the SA heuristic.

V. COMPARISON AND DISCUSSION

We compare the presented research work from Section IV. First, we compile modelling assumptions and problem extensions. Second, we present common scheduling objectives. Then, we investigate problem instances used for evaluations. Finally, we summarize results regarding the scalability of the presented approaches.

A. MODELLING ASSUMPTIONS AND PROBLEM EXTENSIONS

Table 2 compiles important modelling assumptions and problem extensions in the surveyed research works with fixed routings. Table 3 shows the same information for research works about the joint routing problem. In the following section, we compile the contributions to each of these topics.

1) OTHER TRAFFIC

Only five works examine TT and AVB streams simultaneously. Pop et al. [57] present a GRASP heuristic for the handling of AVB streams. The heuristic gets a schedule of TT streams as input and cannot change it. The authors of [125] present a short preview of AVB-aware scheduling, which was later extended in [123]. An adaption of this approach was used in [128]. The authors present an algorithm to assign messages to traffic classes in networks supporting AVB and TSN simultaneously. Feng et al. [83] consider the bandwidth available to AVB and BE traffic in their approach as they consider repeated frame loss which may result in starvation. Berisa et al. [129] use frame preemption and a worst-case end-to-end delay analysis to increase the schedulability of AVB streams. Huang et al. [68] give parameters for AVB streams in an in-vehicle network and include them in their evaluation scenario. Wang et al. [67] consider the joint handling of AVB and TT streams. AVB streams are shaped by CQF. Their objective aims to reduce the influence of non-periodic BE traffic to AVB streams. Some works focus solely on the routing of AVB streams in the presence of scheduled traffic in TSN [124], [127]. Li et al. [58] present a heuristic to configure the CBS in the presence of scheduled traffic.

TABLE 2. Overview of considered problem extensions and restrictions in the literature of scheduling approaches with a fixed routing.

Research work	AVB	BE	Queuing	Fixed GCL length	Reconfiguration	Reliability	Multicast	Task scheduling
Ansah *et al.* [45]	-	-	✓	-	-	-	-	-
Arestova *et al.* [88]	-	-	✓	-	-	-	-	✓
Barzegaran *et al.* [85]	-	-	(✓)	-	-	-	✓	✓
Barzegaran *et al.* [64]	-	(✓)	✓	✓	-	-	-	-
Barzegaran *et al.* [84]	-	-	(✓)	-	-	-	-	✓
Bujosa *et al.* [54]	-	-	✓	-	-	-	✓	-
Chaine *et al.* [53]	-	-	(✓)	-	-	-	-	-
Craciunas *et al.* [29]	-	-	(✓)	-	-	-	-	-
Craciunas *et al.* [69]	-	-	(✓)	-	-	✓	-	-
Dai *et al.* [82]	-	(✓)	-	-	-	-	-	-
Dobrin *et al.* [75]	-	-	✓	-	-	✓	-	-
Duerr *et al.* [28]	-	✓	-	-	-	-	-	-
Farzaneh *et al.* [93]	-	-	-	-	-	-	-	-
Feng *et al.* [70]	✓	✓	(✓)	-	-	✓	-	-
Feng *et al.* [74]	-	✓	✓	-	-	✓	-	-
Feng *et al.* [83]	-	-	(✓)	-	-	-	✓	✓
Gärtner *et al.* [79][80]	-	-	(✓)	-	✓	-	-	-
Gavriluţ *et al.* [125]	✓	-	(✓)	-	-	-	-	-
Gavriluţ *et al.* [128]	✓	✓	(✓)	-	-	-	✓	-
Ginthör *et al.* [91]	-	✓	✓	✓	-	-	-	-
Hellmanns *et al.* [40]	-	(✓)	(✓)	-	-	-	-	-
Houtan *et al.* [62]	-	✓	(✓)	-	-	-	-	-
Huang *et al.* [68]	✓	(✓)	(✓)	-	-	✓	-	-
Jin *et al.* [89]	-	-	-	-	-	-	-	-
Jin *et al.* [30]	-	(✓)	(✓)	✓	-	-	-	-
Kim *et al.* [42]	-	(✓)	✓	-	-	-	-	-
Kim *et al.* [43][44]	-	(✓)	✓	-	-	-	-	-
Lin *et al.* [92]	-	-	-	-	✓	-	-	-
McLean *et al.* [87]	-	-	(✓)	-	-	-	-	✓
Min *et al.* [47]	-	-	-	-	-	-	-	-
Oliver *et al.* [31]	-	(✓)	(✓)	✓	-	-	✓	-
Pang *et al.* [77]	-	-	✓	-	✓	-	✓	-
Park *et al.* [90]	-	-	✓	-	-	✓	-	-
Pei *et al.* [65]	-	✓	✓	-	-	-	-	-
Pop *et al.* [57]	✓	-	(✓)	-	-	-	Only AVB	-
Raagaard *et al.* [76]	-	-	(✓)	-	✓	-	-	-
Reusch *et al.* [52]	-	(✓)	✓	✓	-	-	-	-
Santos *et al.* [59]	-	✓	✓	✓	-	-	✓	-
Steiner [7]	-	-	✓	-	-	-	✓	-
Steiner *et al.* [41]	-	(✓)	(✓)	✓	-	-	✓	-
Vlk *et al.* [48]	-	-	(✓)	-	-	-	-	-
Vlk *et al.* [51]	-	-	✓	-	-	-	-	-
Wang *et al.* [50]	-	(✓)	-	-	-	-	-	-
Wang *et al.* [46]	-	-	✓	✓	-	-	-	-
Wang *et al.* [78]	-	-	✓	-	✓	-	-	-
Wang *et al.* [67]	✓	✓	-	-	-	-	-	-
Yao *et al.* [66]	-	✓	✓	✓	-	-	-	-
Zhang *et al.* [22]	-	-	-	-	-	-	✓	-
Zhou *et al.* [63]	-	-	(✓)	-	-	-	-	-

Other research works handle BE traffic by minimizing the flowspan which yields a large time slot at the end of a schedule exclusively for other traffic, e.g., [28], [40], and [62]. Pei et al. [65] evaluate their approach in the presence of BE traffic and rate constrained traffic. Yao et al. [66] consider the joint scheduling of periodic stream without real-time requirements in their approach. The tables indicate works that do not mention BE traffic, but use objective functions that are beneficial to other traffic or that limit the number of guard bands with (✓). For instance, Oliver et al. [31] limit the number of guard bands indirectly by introducing a fixed number of transmission time windows per egress port.

2) QUEUING

Queuing is a controversial topic in the TSN scheduling literature as non-determinism, e.g., frame loss, may cause serious problems. Some research works do not allow queuing at all, e.g., [28], [71], and [96]. The majority of the algorithms in the literature uses frame isolation introduces by [29]. These works are indicated by (✓) in the Tables 2 and 3. Vlk et al. [51] discuss the effects of isolation constraints. They report results indicating that isolation constraints reduce schedulability significantly. Without isolation, the number of scheduled streams can be increased for all evaluated topologies and problem instance sizes. Additionally, isolation

TABLE 3. Overview of considered problem extensions and restrictions in the literature of joint routing approaches.

Research work	AVB	BE	Queuing	Fixed GCL length	Reconfiguration	Reliability	Multicast	Task scheduling
Alnajim *et al.* [32]	-	✓	✓	-	-	-	-	-
Arestova *et al.* [103]	-	(✓)	(✓)	-	-	-	-	-
Atallah *et al.* [113]	-	-	-	-	-	✓	-	-
Atallah *et al.* [71]	-	-	(✓)	-	-	✓	-	-
Berisa *et al.* [129]	✓	-	✓	-	-	-	-	-
Bhattacharjee *et al.* [145]	-	-	(✓)	-	-	-	-	-
Falk *et al.* [96]	-	-	-	-	-	-	-	-
Falk *et al.* [108]	-	-	-	-	-	-	-	-
Gavriluţ *et al.* [73]	-	-	-	-	-	✓	✓	-
Gavriluţ *et al.* [123]	✓	-	(✓)	-	-	-	-	-
Gong *et al.* [143]	-	-	✓	-	-	-	-	-
He *et al.* [111]	-	-	✓	-	-	-	-	-
Huang *et al.* [98]	-	-	-	-	✓	-	-	-
Kentis *et al.* [106]	-	-	✓	-	-	-	-	-
Li *et al.* [122]	-	-	✓	-	-	✓	✓	-
Li *et al.* [130]	-	✓	✓	✓	-	-	-	-
Li *et al.* [81]	-		-	-	-	-	-	-
Li *et al.* [138]	-	-	-	-	✓	-	-	-
Li *et al.* [133]	-	-	(✓)	-	-	-	✓	-
Mahfouzi *et al.* [95][140]	-	-	✓	-	-	-	-	✓
Nie *et al.* [97]	-	-	-	-	-	-	-	-
Pahlevan *et al.* [104]	-	(✓)	-	-	-	-	-	✓
Pahlevan *et al.* [100]	-	(✓)	-	-	-	-	-	✓
Pozo *et al.* [112]	-	-	-	-	-	✓	✓	-
Reusch *et al.* [55]	-	-	(✓)	-	-	✓	✓	✓
Reusch *et al.* [72]	-	-	-	-	-	✓	-	✓
Schweissguth *et al.* [8]	-	-	-	-	-	-	-	-
Schweissguth *et al.* [132]	-	-	-	-	-	-	✓	-
Smirnov *et al.* [102]	-	✓	-	-	-	-	✓	-
Syed *et al.* [116]	-	-	-	-	-	-	-	-
Syed *et al.* [33]	-	-	-	-	✓	✓	-	-
Syed *et al.* [135]	-	-	-	-	✓	-	-	-
Syed *et al.* [115]	-	-	-	-	-	✓	-	-
Syed *et al.* [117]	-	-	-	-	-	✓	-	-
Vlk *et al.* [109]	-	-	(✓)	-	-	-	-	-
Wang *et al.* [107]	-	-	-	-	-	-	-	-
Xu *et al.* [101]	-	-	(✓)	✓	-	-	-	-
Xu *et al.* [99]	-	-	(✓)	-	-	-	-	-
Yang *et al.* [142]	-	-	✓	-	-	-	-	-
Yang *et al.* [131]	✓	✓	✓	-	-	-	-	-
Yu *et al.* [134]	-	-	(✓)	✓	-	-	✓	-
Yu *et al.* [136]	-	-	(✓)	-	✓	-	✓	-
Zhou *et al.* [118]	-	-	-	-	-	✓	-	-
Zhou *et al.* [114]	-	-	-	-	-	✓	-	-
Zhou *et al.* [121]	-	-	✓	-	-	-	-	-
Zhou *et al.* [141]	-	-	(✓)	-	-	-	-	-

results in larger runtimes compared to scheduling without isolation. In contrast, the differences in end-to-end delays are negligible. However, they propose a different solution to deal with non-determinism, effectively modifying the TAS. Thus, their results are not applicable to current TSN implementations. There are some research works which allow unrestricted queuing, e.g., [59] and [75]. Most of these works do not elaborate on the consequences of unrestricted queuing. In contrast to that, Reusch et al. [52], Barzegaran et al. [64], and Berisa et al. [129] introduced countermeasures for the mentioned consequences. The authors include a worst-case end-to-end delay analysis in their algorithms such that even in the case of non-determinism, deadlines are met.

3) FIXED GCL LENGTH

Most algorithms presented in the literature handle the generation of GCLs indirectly. They schedule transmission offsets of frames at end stations and intermediate bridges. The GCLs are generated by a post-processing after scheduling. This step is only mentioned, and the respective authors do not elaborate on it. Examples for such works are [29], [48], and [55]. However, computing GCLs by a post-processing comes with two drawbacks. First, GCLs have limited size in bridges. Thus, GCLs obtained by a post-processing may not be deployable. Second, the scheduling algorithm cannot include considerations for guard bands. There are some exceptions to this in the literature. Jin et al. [30] present a heuristic to compute schedules with a limited number of GCL entries per egress port. Santos et al. [59] give a detailed SMT model for TSN scheduling which includes the explicit representation of GCLs. Yao et al. [66] limit the number of GCL entries and the maximum queue size in their heuristic, i.e., schedules which do not meet these constraints are not considered valid solutions. Some works limit the number of GCL entries indirectly by introducing transmission windows for egress ports. Streams are mapped to these transmission windows and their number is fixed before scheduling. Examples of such works are [31], [41], and [52]. All schedules for no-wait scheduling can be deployed with a fixed number of GCL entries. As no queuing delay is allowed, frames cannot be scheduled to wait at closed gates. Such a schedule can be deployed by opening all gates for TT traffic at the start of a hyperperiod and never closing them.

4) RECONFIGURATION

In some scenarios it may be infeasible to compute new schedules every time a stream should be integrated into or removed from an existing schedule. For instance, automotive scenarios may include ad-hoc connections between cars and infrastructure. Computing new schedules every time a new stream is added may take too much time, even with heuristic algorithms. Syed et al. [33], [135] consider reconfiguration in such automotive scenarios. Additionally, they also present preliminary work about computing schedules suitable for later reconfiguration in [116]. Raagaard et al. [76] present a heuristic to add streams to an existing schedule. When the heuristic fails, they assign the new stream to other egress queues which were unused before. Pang et al. [77] present work about deploying an updated schedule to a network already executing another schedule. Their approach allows updating the schedule without frame loss or new streams interfering with the old schedule. Another use case for reconfiguration is the reallocation of tasks sending and receiving TT streams. Yu et al. [136] use virtual machines as end stations in their model. These virtual machines may be migrated from one physical device to another, which requires updating schedules and routings. A similar example for reconfiguration is presented in [138]. The authors propose an approach for updating a schedule in case of a permanent end station failure. Schedules and routings must be updated in this case as in [136]. Gärtner et al. [79] introduce a measure for schedule flexibility which also considers deadlines. They use this measure to update schedules in a beneficial way for future updates. Lin et al. [92] show in their evaluations that aligning frame transmissions to the greatest common divisor results in schedules that are suitable for adding streams later.

TABLE 4. Fault models in research works dedicated to reliability.

Fault Model	Research work
Permanent link failure	Atallah *et al.* [113], Atallah *et al.* [71], Gavriluţ *et al.* [73], Pozo *et al.* [112], Reusch *et al.* [55], Reusch *et al.* [72], Syed *et al.* [33], Syed *et al.* [117]
Frame loss	Atallah *et al.* [113], Atallah *et al.* [71], Dobrin *et al.* [75], Feng *et al.* [70] [74], Gavriluţ *et al.* [73], Huang *et al.* [68], Li *et al.* [122], Park *et al.* [90], Reusch *et al.* [55], Reusch *et al.* [72], Syed *et al.* [33], Syed *et al.* [115], Syed *et al.* [117], Zhou *et al.* [114]
Clock drift	Craciunas *et al.* [69]
Hardware bugs	Zhou *et al.* [118]

5) RELIABILITY

Table 4 compiles fault models used in the literature. The listed research works construct schedules robust in the respective fault model. Computing schedules robust against frame loss is the most common kind of reliability in the TSN scheduling literature. Park et al. [77] handle frame loss by allowing the retransmission of frames. Schedules for such a scenario must schedule enough time between frame transmissions such that retransmissions do not interfere with other frames. Another way to deal with frame loss is proposed by Feng et al. [70], [74]. In contrast to [77], they do not use retransmissions, but they schedule redundant copies of the same stream over the same path. Zhou et al. [114] approximate the probability of frame loss in a joint routing and scheduling model. Redundant copies of streams are routed over not necessarily disjoint paths to reduce the probability of frame loss. Robustness against permanent single link failures are also covered in several works. There are two approaches in the TSN scheduling literature to handles such failures. First, redundant copies of streams are scheduled and routed over link-disjoint paths before a link failure arises. Examples for such works are [33], [71], [73], and [113]. Huang et al. [68] and Syed et al. [115] use Frame Replication and Elimination for Reliability [17] to implement this approach. Second, streams can be rescheduled and rerouted after a link failure occurred. Pozo et al. [112] present a heuristic for fast rescheduling and rerouting in this case. We remark that all works about computing schedules robust against permanent link failures are also robust against frame loss. Both countermeasures against link failures are also effective against frame loss. Another kind of reliability is considered in [69]. The authors compute schedules robust against clock drift, i.e., clocks of different devices running not with the same speed. They introduce gaps between frame transmissions such that the maximum possible clock drift does not affect other frame transmissions. Unknown hardware bugs or deviations from TSN standards are treated by [118]. The proposed algorithm selects expensive bridges with higher certification for paths of streams with higher safety requirements. In contrast to all other mentioned works with reliability, Syed et al. [117] use an encoding scheme to reconstruct lost frames. The XORed data of two frames is transmitted over a path disjoint to the paths of both frames.

6) MULTICAST

Every algorithm in literature can be used for multicast streams, as a multicast stream can be substituted by a set of unicast streams. However, the number of streams negatively affects the solving time for a problem instance. Tables 2 and 3 indicate multicast support only for works which include some considerations for the efficient integration of multicast streams without introducing a set of new streams. Most such works handle multicast streams by scheduling only a single frame per link, regardless of the number of consecutive links in the multicast tree of the respective stream. Examples for such works are [31], [41], [59], [83], [85], and [128]. An analysis of the joint routing and scheduling problem with multicast streams is presented in [132]. Yu et al. [136] compute routings and schedules for multicast streams such that migrating a virtual machine sending or receiving TT streams can be done easily. Some works allow multicast streams, but do not elaborate on the implementation details, e.g., [22].

7) TASK SCHEDULING

Only a few research works are concerned with the joint scheduling of streams and tasks. Some of them have integrated dependencies between streams and tasks, i.e., tasks can only be scheduled after some stream has arrived. Such works are presented in [83], [84], [85], [87], [88], and [104]. Other works focus on the scheduling of tasks which produce TT streams while considering safety and security considerations. Preliminary results for this scenario are presented in [72] and extended in [55].

8) SECURITY CONSIDERATIONS

While many works focus on the reliability of data transmissions, security aspects were mostly ignored so far. An exception to this are the works of Reusch et al. [55], [72]. The authors identified the problem of replay and impersonation attacks. However, security aspects are not covered by current TSN standards. The authors propose to use the TESLA protocol [146] to mitigate these problems. The additional messages for key exchanges and the additional tasks for verification and key management are considered during scheduling.

B. SCHEDULING OBJECTIVE

Objective functions are used to measure the quality of solutions and to compare them. We discuss common objectives from the literature and classify research works by their objective. Table 5 shows which research work features which kind of objective.

TABLE 5. Categorization of research works based on optimization objectives.

Objective Category	Research work
No objective	Alnajim *et al.* [32], Ansah *et al.* [45], Atallah *et al.* [113], Atallah *et al.* [71], Bujosa *et al.* [54], Dai *et al.* [82], Dobrin *et al.* [75], Falk *et al.* [96][108], Farzaneh *et al.* [93], Gavriluţ *et al.* [128], He *et al.* [111], Kim *et al.* [42], Li *et al.* [130], Li *et al.* [81], Mahfouzi *et al.* [95], Mahfouzi *et al.* [140], Raagaard *et al.* [76], Santos *et al.* [59], Steiner [7], Steiner *et al.* [41], Syed *et al.* [135] [33] [117], Vlk *et al.* [48], Wang *et al.* [78], Wang *et al.* [46], Xu *et al.* [99], Yao *et al.* [66], Zhou *et al.* [63], Zhou *et al.* [114], Zhou *et al.* [121]
Latency and Jitter	Arestova *et al.* [103] [88], Barzegaran *et al.* [85], Barzegaran *et al.* [84], [100], Dürr *et al.* [28], Feng *et al.* [70], Gärtner *et al.* [79][80], Hellmanns *et al.* [40], Houtan *et al.* [62], Huang *et al.* [98], Huang *et al.* [68], Kim *et al.* [43][44], McLean *et al.* [87], Nie *et al.* [97], Oliver *et al.* [31], Pahlevan *et al.* [104], Pang *et al.* [77], Pei *et al.* [65], Schweissguth *et al.* [8] [132], Vlk *et al.* [51], Wang *et al.* [50], Yang *et al.* [131], Zhang *et al.* [22], Zhou *et al.* [141]
Queuing	Craciunas *et al.* [29], Feng *et al.* [74], Gavriluţ *et al.* [125], Pop *et al.* [57], Vlk *et al.* [51], Vlk *et al.* [109]
Other Traffic	Barzegaran *et al.* [64], Berisa *et al.* [129], Gavriluţ *et al.* [125] [123], Houtan *et al.* [62], Reusch *et al.* [52], Smirnov *et al.* [102], Wang *et al.* [67]
Routing	Li *et al.* [122], Li *et al.* [138], Li *et al.* [133], Schweissguth *et al.* [132], Wang *et al.* [107], Yu *et al.* [134], Yu *et al.* [136]
Topology Synthesis	Gavriluţ *et al.* [73], Reusch *et al.* [55], Xu *et al.* [101], Zhou *et al.* [118]
Reliability	Craciunas *et al.* [69], Park *et al.* [90], Pozo *et al.* [112]
GCL Synthesis	Kentis *et al.* [106], Jin *et al.* [30]
Other	Bhattacharjee *et al.* [145], Chaine *et al.* [53], Feng *et al.* [83], Ginthör *et al.* [91], Gong *et al.* [143], Jin *et al.* [89], Lin *et al.* [92], Min *et al.* [47], Syed *et al.* [116], Syed *et al.* [115], Yang *et al.* [142], Yang *et al.* [142]

1) NO OBJECTIVE

Many research works have no scheduling objective and only try to find some schedule which fulfills all constraints, e.g., [76], [96], [108], and [111]. We note that many SMT approaches feature no objective [7], [41], [59], [93], [95], [140]. In contrast to ILP solving, SMT solvers were not originally designed for optimization. Therefore, many SMT approaches focus on finding a feasible schedule.

2) LATENCY AND JITTER

TSN and the TAS were designed for traffic with hard real-time requirements. Therefore, latency and jitter of streams are interesting properties of schedules. Objective functions including them are the most common kind of objectives in TSN schedule optimization. Oliver et al. [31] and Barzegaran et al. [84] minimize the per-stream jitter. Minimizing the flowspan, i.e., the time all TT stream arrive at their destination, is a common objective. Examples of approaches using this objective include [28], [40], [50], [62], [68], [100], [103], and [104]. A related but different objective is the minimization of end-to-end delays of TT streams [8], [51], [65], [77], [79], [131], [132]. Kim et al. [43], [44] minimize multiple objectives weighted by constant factors. They take end-to-end delays, jitter, and bandwidth occupation into account. Barzegaran et al. [85] use a combination of jitter and end-to-end latency as measure of schedule quality. Nie et al. [97] minimize end-to-end latency and transmission offsets simultaneously. We remark that these objectives are not competing, as opposed to most multi-criterion problems. Minimization of transmission offsets is also pursued by [22] and [98] which is related but not equal to flowspan or end-to-end latency minimization. Zhou et al. [141] use a combination of jitter, end-to-end delays, number of scheduled streams, and link utilization.

3) QUEUING

Research works which apply isolation constraints for queuing often use more than one queue per egress port for scheduled traffic. In that way, they are able to schedule more streams as isolation only concern streams in the same egress queue. The assignment of streams to egress queues per egress port is a degree of freedom in the respective scheduling problems. Therefore, they try to minimize the number of queues reserved for TT streams per egress port, as the remaining queues are available for other traffic. Examples for such works include [29], [51], [57], [74], [109], and [125].

4) OTHER TRAFFIC

The schedule of TT streams has an influence on the Quality of Service for other traffic classes. Current approaches for scheduling in TSN focus on AVB traffic and BE traffic. For the joint scheduling of TT and AVB streams, Gavriluţ et al. [123], [125] minimize the tardiness of AVB streams as their deadlines are considered to be not strict. Another objective related to AVB streams is used in [129]. The presented heuristic has the objective to schedule as many AVB streams as possible. Yang et al. [131] minimize the the weighted sum of stream latencies of scheduled traffic, AVB, and BE streams. Wand et al. [67] use CQF to shape AVB streams in their approach. The objective function aims to load balance the AVB frame transmissions between the time slots of the CQF mechanism. This reduces the probability that non-periodic BE traffic overloads such a slot. The authors of [52] minimize the occupation percentage of egress ports, i.e., the percentage of a hyperperiod with no active transmission window for TT traffic. The rational of this is that low occupation corresponds to long and frequent time intervals available to other traffic. A similar objective is used in [64] as the authors minimize the average bandwidth occupied by transmission windows for scheduled traffic. Smirnov et al. [102] use a multi-criterion objective for joint routing and scheduling. They reduce the influence of scheduled traffic to other traffic, and simultaneously minimize the number of GCL entries needed to deploy a schedule. The work in [62] focuses on comparing the influence of different objective functions to the QoS of BE traffic. They propose minimization and maximization of frame offsets, hoping that grouping frames together increases the QoS. Additionally, they also suggest two objectives which maximize the gaps between consecutive frame transmissions on a link. They assume that starvation of other traffic classes is reduced in this way.

5) ROUTING

Research works about joint routing and scheduling often consider the quality of the routing in their objective. All of them have in common that the length of the paths is minimized. This is reasonable as longer paths correspond to higher link utilizations, end-to-end latencies, and harder scheduling instances. Schweissguth et al. [132] propose a multi-objective optimization for joint routing and scheduling. First, routing and schedule with minimized path lengths are computed. The obtained path lengths are used as maximum path lengths per stream in a second run. The second run minimizes end-to-end latencies. The joint routing approach of [107] minimizes the number of links in the routing. Li et al. [138] simultaneously minimize the path lengths and a measure for scheduling conflicts of streams routed over the same link. Yu et al. [134] schedule and route streams one after another. They minimize a weighted sum of the number of links used for the currently scheduled stream, and the bandwidth utilization. Li et al. [133] simultaneously minimize path lengths in the routing, and the flowspan. Yu et al. [136] consider the migration of sources of TT streams. They minimize the maximum distance from all possible source nodes of a stream to all destination nodes in a multicast tree. Li et al. [122] maximize the number of streams which are scheduled and routed, and also try to minimize the maximum link load as a secondary objective.

6) TOPOLOGY SYNTHESIS

In addition to joint routing and scheduling, some works also construct the network topology. TSN bridges are expensive, and thus such objectives always include costs for bridges. Gavriluţ et al. [73] minimize multiple objectives weighted by constant factors. The first objective is the tardiness of TT streams to guide their GRASP heuristic to solutions with no deadline misses. The second objective is topology costs. A similar objective is used in [55]. The weighted sum of routing and schedule costs is minimized. Routing costs constitute of overlap penalties for redundant paths and path lengths. Schedule costs constitute of punishments for not schedulable streams and stream latencies. Another approach which minimizes topology costs is proposed in [118]. Selecting bridges from a library is part of the presented problem, which imposes costs for bridges and additional costs when multiple vendors are used. Xu et al. [101] minimize the number of bridges needed to schedule and route all streams such that the utilization is maximized.

7) RELIABILITY

Reliability requirements can be ensured by constraining the set of feasible solutions. However, some works choose to maximize reliability for their respective fault model. Pozo et al. [112] maximize the idle times of links and frames, as such schedules are easier to repair upon link failure. Craciunas et al. [69] maximize the allowed out-of-sync clock drift to cope with synchronization problems and maximize robustness against clock drift. Park et. al. [90] maximize the number of times a frame can be retransmitted without missing its deadline, as they include preemption in their model.

8) GCL SYNTHESIS

TSN bridges do not have an unlimited number of GCL entries per egress port. The minimization of GCL entries is considered by [30]. The reason for this is that the authors propose an incremental approach and the overall number of needed GCL entries is not known in advance. Reducing the number of gate events also reduces the number of guard bands which is beneficial for BE traffic. Kentis et al. [106] minimize the GCL schedule duration. However, it is not clear why schedule duration matters, as the limiting factor in TSN hardware is the number of GCL entries.

9) OTHERS

Some research works use a problem specific objective not comparable to other works. We present them for the sake of completeness. The authors of [89] minimize the number of frames as they propose a joint approach for scheduling and message fragmentation. Syed et al. [115] and [116] use a modelling specific objective which is related to load balancing of ports in an in-vehicle architecture with one central processing unit. Ginthür et al. [91] minimize the wasted bandwidth for different link layer technologies, i.e., Ethernet and 5G links. Feng et al. [83] minimize the response time of tasks which may be dependent on streams as they consider the joint scheduling of streams and tasks. Chaine et al. [110] maximize the length of transmission time windows of streams at their respective talkers such that latency and jitter requirements are met. This is the only work in TSN scheduling which employs a quadratic objective function. Bhattacharjee et al. [145] employ a multi-criterion objective. Their first objective is to minimize the maximum load across all servers as the considered problem includes the placement of talker applications. The second objective is to minimize the average hop count of all streams. Lin et al. [92] present a heuristic for incremental scheduling that aims to maximize the probability that more streams can be added later. This is required in industrial use cases as turning off machines to deploy a new schedule may be expensive. Yang et al. [142] state that they maximize the number of scheduled streams while minimizing the link occupancy rate. However, this rate is not defined in the published magazine article. Gong et al. [143] also maximize the occupancy rate while minimizing the maximum link utilization. They define the occupancy rate as the fraction of bandwidth reserved for scheduled traffic actually used for transmissions. Min et al. [47] compare metaheuristics by maximizing the number of scheduled stream for the same problem instance.

TABLE 6. Overview of investigated problem instances in the literature of the scheduling problem with fixed routing.

Research work	Topology	ES	Bridges	TT streams
Arestova *et al.* [88]	Mesh, ring	N/A	5 – 30	N/A
Barzegaran *et al.* [85]	Various	6 – 20	2 – 20	8 – 27
Barzegaran *et al.* [64]	Various	3 – 31	2 – 15	7 – 137
Barzegaran *et al.* [84]	Various	5 – 20	2 – 20	8 – 27
Bujosa *et al.* [54]	Line	3 – 9	1 – 3	N/A
Chaine *et al.* [53]	Line, spacecraft	2 – 31	4 – 15	15 – 304
Craciunas *et al.* [29]	N/A	3–7	1–5	5–1000
Craciunas *et al.* [69]	Tree	16	7	96
Dai *et al.* [82]	Link	N/A	N/A	3
Dürr *et al.* [28]	ER, RRG, BA	24–100	5–20	30–1500
Farzaneh *et al.* [93]	Snowflake	12	2	14–100
Feng *et al.* [70]	N/A	2 – 8	2 – 5	2 – 16
Feng *et al.* [74]	N/A	5 – 8	3 – 5	N/A
Feng *et al.* [83]	N/A	50	10	11
Gärtner *et al.* [79][80]	Line, machine	2 – 54	2 – 15	1 – 104
Gavriluţ *et al.* [125]	Star, Snowflake	3–32	1–18	4–35
Gavriluţ *et al.* [128]	Various	7–31	1–15	20–186
Ginthör *et al.* [91]	N/A	N/A	N/A	10
Hellmanns *et al.* [40]	Ring of rings	100–2500		10–2000
Hellmanns *et al.* [144]	Ring of rings	N/A	N/A	50–500
Houtan *et al.* [62]	Snowflake	6	2	10
Huang *et al.* [68]	Automotive	33	14	20 – 480
Jin *et al.* [89]	N/A	5–30	5–30	10–60
Jin *et al.* [30]	N/A	N/A	6–20	10–10000
Kim *et al.* [42]	Various	7	5	6
Kim *et al.* [43][44]	Automotive	17	4	27
Li *et al.* [81]	Various	30 – 100	7 – 20	3 – 100
Lin *et al.* [92]	Tree, mesh, ring	4 – 16	1 – 4	~ 48 – 223
McLean *et al.* [87]	Tree, mesh, ring	4 – 432	2 – 43	16 – 1728
Min *et al.* [47]	Real-world	14		300
Oliver *et al.* [31]	Line	50	10	10–50
Pang *et al.* [77]	In-train, spacecraft	31 – 54	13 – 31	N/A
Park *et al.* [90]	Automotive	5 – 20	3 – 7	100 – 500
Pei *et al.* [65]	Spacecraft	31	15	20
Pop *et al.* [57]	N/A	3–5	1–2	3–5
Raagaard *et al.* [76]	N/A	4–402		15–290
Reusch *et al.* [72]	Various	4–32	1–16	N/A
Reusch *et al.* [52]	Snowflake	3–256	1–146	14–316
Santos *et al.* [59]	N/A	50	10	1 – 10
Steiner [7]	Star, tree, snowflake	N/A	N/A	100 – 1000
Steiner *et al.* [41]	N/A	50	10	10–50
Vlk *et al.* [48]	Ring of lines, ring of trees, spacecraft	~ 20 – 1700	~ 20 – 300	1500 – 12000
Vlk *et al.* [51]	Mesh, ring, tree	4 – 16	1 – 4	6 – 450
Wang *et al.* [50]	N/A	4 – 10	3 – 9	10 – 100
Wang *et al.* [46]	Automotive	4	1	3
Wang *et al.* [78]	Mesh	15 – 16	6 – 8	N/A
Wang *et al.* [67]	Line, ring, tree	N/A	5 – 200	N/A
Yao *et al.* [66]	N/A	N/A	N/A	10 – 20
Zhou *et al.* [63]	Automotive	5	16	N/A
Zhang *et al.* [22]	Tree, line, tree	6 – 8		200 – 1200

TABLE 7. Overview of investigated problem instances in the literature of the joint routing problem.

Research work	Topology	ES	Bridges	TT streams
Alnajim *et al.* [32]	N/A	30–150	10–50	100–1500
Arestova *et al.* [103]	N/A	50	10	10 – 100
Atallah *et al.* [113]	N/A	6 – 24	N/A	30 – 600
Atallah *et al.* [71]	N/A	N/A	3–21	20–60
Berisa *et al.* [129]	Ring, full mesh, spacecraft	13 – 31	4 – 15	222
Bhattacharjee *et al.* [145]	Ring, BA, ER	46 – 1024	24 – 512	90 – 1845
Falk *et al.* [96]	Line, ring, BA, ER	5–36		2–30
Falk *et al.* [108]	Ring w/ k neighbors	50–400		50–150
Gavriluţ *et al.* [73]	N/A	4–20	N/A	4–38
Gavriluţ *et al.* [123]	Various	3–256	2–146	4–427
Gong *et al.* [143]	N/A	N/A	N/A	15 – 180
He *et al.* [111]	ER, RRG, BA	20		25 – 200
Huang *et al.* [98]	Spacecraft, ER	N/A	N/A	500 – 2000
Kentis *et al.* [106]	Ring, mesh	12	12	20–52
Li *et al.* [122]	ER, RRG, BA	10	10	10 – 100
Li *et al.* [130]	Mesh, automotive	15 – 105	3 – 21	2 – 4000
Li *et al.* [138]	Real-world	31	13	100 – 300
Li *et al.* [133]	N/A	39	16	10 – 40
Mahfouzi *et al.* [95][140]	ER, Automotive	20	10–45	19–106
Nie *et al.* [97]	Ring, mesh	11 – 14	4 – 15	10 – 40
Pahlevan *et al.* [104]	Grid, ring	27–45	9	30–40
Pahlevan *et al.* [100]	Grid, ring	50	10	60–100
Pozo *et al.* [112]	Synthetic	6–8	3–8	10–50
Reusch *et al.* [55]	Various	4 – 128	2–64	2 – 144
Reusch *et al.* [72]	Various	4–32	1–16	N/A
Schweissguth *et al.* [8]	Ring, mesh	13	N/A	60
Schweissguth *et al.* [132]	Ring, mesh	12	12	25 – 40
Smirnov *et al.* [102]	N/A	N/A	N/A	5 – 75
Syed *et al.* [116]	Automotive	N/A	N/A	20–90
Syed *et al.* [135]	Automotive	N/A	N/A	100–500
Syed *et al.* [33]	Automotive	N/A	N/A	100–500
Syed *et al.* [115]	Automotive	N/A	N/A	20–90
Syed *et al.* [117]	Automotive	N/A	N/A	112
Vlk *et al.* [109]	Ring, mesh	12 – 48	12 – 48	10 – 300
Wang *et al.* [107]	N/A	25	5	10–100
Xu *et al.* [101]	Mesh	5	1 – 23	4 – 12
Xu *et al.* [99]	Mesh, ring	N/A	7–15	40 – 80
Yang *et al.* [142]	N/A	2	N/A	45 – 180
Yang *et al.* [131]	Star, tree, BA	N/A	8 – 18	150 – 900
Yu *et al.* [134]	Mesh	72	24	1 – 500
Yu *et al.* [136]	BA	72	24	10 – 350
Zhou *et al.* [118]	ER, automotive	N/A	6–10	50 – 240
Zhou *et al.* [114]	ER, automotive	16	6–10	50–380
Zhou *et al.* [121]	ER, real-world	16	4–64	30 – 220
Zhou *et al.* [141]	Mesh	≥ 4	3 – 15	50 – 650

C. PROBLEM INSTANCES

Before we describe evaluation results, we describe the problem instances used for evaluations in the literature. We present an overview of used network topologies, network sizes, and numbers of streams. Tables 6 and 7 compile this information about the problem instances used for evaluations with fixed routing and joint routing, respectively.

Unfortunately, some research works do not elaborate on the used topologies, which makes assessing and comparing the results to other works harder. Most research works only use synthetic test cases. Ring topologies are commonly used in evaluations, e.g., in [8], [40], [67], [87], [88], [96], [97], [100], [101], [104], [108], [109], and [132]. Hellmanns et al. [40] argue that rings are a common topology in real-world industrial facilities. Other systematic topologies used include line [31], [67], [96], grid [100], [104], and snowflake-like [52], [62], [93], [125] networks. Various randomly generated topologies are also used in evaluations. Erdós-Rényi graphs (ER) are the most common ones [28], [96], [111], [114] [118], [121], [122], but Barabási-Albert graphs (BA) [28], [96], [111], [122] [134] and random regular graphs (RRG) [28], [111], [122] are also used. A few research works features evaluations with real-world topologies. Syed et al. [33], [115], [116], [117], [135] use a real-world automotive architecture for their evaluations. A large automotive architecture including stream parameters is discussed in [68]. Other automotive architectures are used by Kim et al. [43], [44], Li et al. [130], Mahfouzi et al. [95], [140], and Wang et al. [46]. Zhou et al. [114], [118] conclude their evaluations by investigating a real-world example from General Motors. The authors of [73] also use a real-world problem instance from General Motors. However, this instance is only a set of streams without topology. Min et al. [47] use the network topology of the National Science Foundation of the United States of America. Barzegaran et al. [64] presents evaluations with real-world test cases from General Motors and a real-world spacecraft. Pang et al. [77] evaluate an algorithm for schedule updates in a real-world in-train network and a spacecraft. Similarly, the authors of [79] evaluate their algorithm for schedule reconfigurations with the topology of a not specified machine. Vlk et al. [48], Chaine et al. [53], Huang et al. [98], Gavriluţ et al. [128], and Berisa et al. [129], perform evaluations with a real-world spacecraft topology.

All research works concerned with synthetic test cases use randomly generated streams. Sources and destinations of these streams are selected uniformly from the sets of talkers and listeners in the respective topology. The number of streams varies considerably between different research works. It ranges from 2 streams in the smallest instance of Falk et al. [96] to up to 10812 streams in the largest instance of Vlk et al. [48]. All works, which describe the placement of deadlines, place them at the end of the respective stream's period. No research work allows deadlines to be after the end of the hyperperiod a frame was sent. Stream periods range from 32 μs in [30] to 500 ms in [29]. All research works assume transmission rates of either 100 Mb/s or 1 Gb/s per egress port.

D. SCALABILITY

Scheduling in TSN is known to be NP-complete. Therefore, solving times and sizes of feasible problem instances matter. Almost all research works about TSN scheduling include or even focus on evaluating the scalability of the respective proposed approach. These evaluations measure the solving times for selected problem instances. Tables 8 and 9 compile the reported runtimes needed to solve the largest problem instance for which a schedule was found in the respective research work. Tables 10 and 11 report the same results for research works which feature the joint computation of schedules and routings. We divided results in separate tables for exact and heuristic algorithms for better comparability. In cases where it was not clear which problem instance can be considered as the largest one, we used the number of streams as tie-breaker. This is justified by several research works surveyed in this paper, e.g., in [96]. The tables are meant to show general tendencies and improvements, not to suggest one approach over the other. Caution is needed when interpreting the tables. It shows the reported times after which an algorithm terminated, not the time until a first valid schedule was obtained, as almost all papers do not report this time. This is a systematic disadvantage of exact approaches as they only terminate when the optimal solution is found or some timeout is reached, while heuristic algorithms may terminate much earlier with suboptimal solutions. Some research works deal with more parameters than the size of the network and the number of streams, e.g., Oliver et al. [31] present evaluations about the influence of the number of transmission windows per egress port to scalability. Other works handle problem extensions, e.g., AVB or task scheduling. Approximations are given when results are not stated in the text and had to be estimated by the presented figures. Ranges are given when multiple instances are considered to be the largest. We identified two tendencies with respect to solving times.

First, heuristic approaches can handle larger instances than approaches with exact solution methods. While the number of network nodes is approximately in the same range, heuristic algorithms can schedule problem instances with more streams compared to exact approaches. Typical numbers of streams in exact approaches are less than 100, e.g., in [8], [69], and [96]. However, there are some notable exceptions. Craciunas et al. [29] present an incremental scheduling algorithm with backtracking, which scheduled instances with 1000 streams in their evaluations. Later works present incremental approaches which were able to schedule as many as 2000 streams [98]. Oliver et al. [31] assigned streams to transmission windows of egress ports and report solved instances with 750 streams. Heuristic approaches were able to schedule instances with more than 10000 streams, e.g., [48] and [30].

Second, exact approaches which solve the joint routing and scheduling problem can only handle instances with smaller numbers of nodes compared to approaches solely for scheduling. Typical networks in evaluations of joint routing algorithms contain less than 50 nodes [71], [72], [73]. This is due to the solution space growing heavily with an increased number of possible paths per stream. However, there are approaches able to compute routings and schedules for problem instances with up to 96 nodes [109], [134]. Most networks in the literature of scheduling with a fixed routing contain less than 96 nodes. The range of the number of

TABLE 8. Overview of solving times of the respective largest reported problem instance for which a schedule was found. Only research works with exact approach and fixed routing are included for comparability.

Research work	Solution approach	Nodes	TT streams	Runtime (s)
Barzegaran *et al.* [64]	CP	120	500	N/A
Barzegaran *et al.* [84]	CP	40	27	2563
Chaine *et al.* [53]	ILP	46	304	~ 240
Craciunas *et al.* [29]	SMT	12	1000	< 18000
Craciunas *et al.* [69]	SMT	23	96	~ 0.343 – 0.437
Dai *et al.* [82]	CP	N/A	3	N/A
Farzaneh *et al.* [93]	SMT	14	100	< 240
Feng *et al.* [70]	SMT	13	16	N/A
Feng *et al.* [74]	SMT	13	N/A	~ 900
Feng *et al.* [83]	SMT	60	11	384 – 694
Ginthör *et al.* [91]	CP	N/A	10	N/A
Houtan *et al.* [62]	SMT	8	10	0.37–1153.52
Jin *et al.* [89]	SMT	4	4	< 600
Jin *et al.* [30]	SMT	6	50	660–1080
Li *et al.* [81]	SMT	120	100	~ 320 – 450
Nie *et al.* [97]	ILP	26	40	~ 0 – 10
Oliver *et al.* [31]	SMT	20	750	~ 600
Pang *et al.* [77]	ILP	86	N/A	~ 620 – 900
Pop *et al.* [57]	ILP	7	5	80.19
Santos *et al.* [59]	SMT	60	10	< 288000
Steiner [7]	SMT	N/A	1000	~ 180 – 1140
Steiner *et al.* [41]	SMT	60	50	~ 0.01 – 100
Vlk *et al.* [51]	ILP	20	414	≤ 600
Zhou *et al.* [63]	SMT	21	N/A	N/A

TABLE 9. Overview of solving times of the respective largest reported problem instance for which a schedule was found. Only research works with heuristic approach and fixed routing are included for comparability.

Research work	Solution approach	Nodes	TT streams	Runtime (s)
Arestova *et al.* [88]	Heuristic	> 30	N/A	105.4
Atallah *et al.* [113]	Heuristic	24	600	~ 8
Barzegaran *et al.* [85]	CP + heuristic	40	27	3553 – 9153
Barzegaran *et al.* [84]	CP + heuristic	40	27	161
Bujosa *et al.* [54]	Heuristic	12	N/A	< 0.01
Dürr *et al.* [28]	Tabu search	120	1500	11520
Gärtner *et al.* [79][80]	Heuristic	69	104	~ 0.1
Gavriluţ *et al.* [125]	GRASP	50	35	612.8
Gavriluţ *et al.* [128]	GRASP	46	186	750
Hellmanns *et al.* [40]	Tabu search	2500	2000	~ 4400
Huang *et al.* [68]	Heuristic	47	480	~ 2700
Jin *et al.* [30]	Heuristic	20	10000	~ 5100
Kim *et al.* [42]	Heuristic	12	6	N/A
Kim *et al.* [43][44]	GA	21	27	12300
Lin *et al.* [92]	Heuristic	20	~ 223	N/A
McLean *et al.* [87]	Heuristic	475	1728	~ 10000
Park *et al.* [90]	GA	< 27	500	N/A
Pei *et al.* [65]	Heuristic	46	20	N/A
Raagaard *et al.* [76]	Heuristic	402	290	20–54
Reusch *et al.* [52]	Heuristic	402	316	10.52
Vlk *et al.* [48]	Heuristic	2000	10812	~ 1000
Wang *et al.* [50]	Machine learning	19	100	~ 400
Wang *et al.* [46]	Heuristic	5	3	< 1
Wang *et al.* [78]	Heuristic	23	N/A	N/A
Wang *et al.* [67]	Heuristic	16	200	~ 1000
Yao *et al.* [66]	Heuristic	N/A	20	~ 2
Zhang *et al.* [22]	Heuristic	8	1200	~ 11

TABLE 10. Overview of solving times of the respective largest reported problem instance for which a schedule was found. Only research works with exact approach and joint routing are included for comparability.

Research work	Solution Approach	Nodes	TT streams	Runtime (s)
Atallah *et al.* [71]	ILP	14	60	~ 100
Falk *et al.* [96]	ILP	8	30	~ 170 – 1580
Gavriluţ *et al.* [73]	CP	20	38	172800
Hellmanns *et al.* [144]	ILP	N/A	N/A	50–500
Huang *et al.* [98]	ILP	44	2000	1620
Li *et al.* [133]	ILP	55	40	~ 80
Mahfouzi *et al.* [95][140]	SMT	65	45	~ 21
Pozo *et al.* [112]	ILP	16	50	2–85
Reusch *et al.* [55]	CP	48	33	1500
Reusch *et al.* [72]	CP	48	N/A	~ 1800
Schweissguth *et al.* [8]	ILP	24	52	1284.53
Schweissguth *et al.* [132]	ILP	24	~ 46 – 60	N/A
Smirnov *et al.* [102]	PBO	N/A	75	~ 42 – 78
Syed *et al.* [116]	ILP	N/A	90	~ 21000
Vlk *et al.* [109]	CP	96	300	~ 100
Xu *et al.* [101]	SMT	24	12	~ 144000
Xu *et al.* [99]	SMT	12	80	~ 850
Yu *et al.* [134]	ILP	96	450	N/A
Zhou *et al.* [118]	SMT	10	240	~ 3500 – 4000
Zhou *et al.* [114]	SMT	10	380	~ 2700
Zhou *et al.* [121]	SMT	24	220	~ 55 – 440

TABLE 11. Overview of solving times of the respective largest reported problem instance for which a schedule was found. Only research works with heuristic approach and joint routing are included for comparability.

Research work	Solution Approach	Nodes	TT streams	Runtime (s)
Alnajim *et al.* [32]	Heuristic	200	1500	2718
Arestova *et al.* [103]	Genetic algorithm	60	100	~ 470
Berisa *et al.* [129]	Heuristic	46	222	602 – 7090
Bhattacharjee *et al.* [145]	SA	1536	1845	~ 1800
Falk *et al.* [108]	Heuristic	400	400	~ 6000 – 11000
Gavriluţ *et al.* [123]	GRASP	402	427	534.6
Gavriluţ *et al.* [73]	GRASP	20	38	558
	Heuristic	20	38	130
Gong *et al.* [143]	Tabu search	N/A	180	~ 11160
He *et al.* [111]	Machine learning	20	200	7
Kentis *et al.* [106]	Heuristic	≥ 13	60	N/A
Li *et al.* [122]	Heuristic	20	100	N/A
Li *et al.* [130]	Heuristic	126	4000	0.437 – 0.88
Li *et al.* [138]	ILP + heuristic	44	300	~ 0.3
Pahlevan *et al.* [104]	List scheduler	45	40	0.014
	Genetic algorithm	45	40	56.75
Pahlevan *et al.* [100]	List scheduler	50	100	0.103
	Heuristic	50	100	1.58
Reusch *et al.* [55]	SA + list scheduler	192	144	1200
Syed *et al.* [33]	Heuristic	N/A	500	0.41
Syed *et al.* [135]	Heuristic	N/A	500	0.170
Syed *et al.* [115]	Heuristic	N/A	90	~ 342
Syed *et al.* [117]	Heuristic	N/A	90	~ 8 – 10
Wang *et al.* [107]	Heuristic	30	100	72
Yang *et al.* [142]	Tabu search	N/A	180	~ 11160
Yang *et al.* [131]	Machine learning	18	900	N/A
Yu *et al.* [136]	Heuristic	96	350	N/A
Zhou *et al.* [141]	Heuristic	N/A	650	~ 2700

streams is approximately the same for approaches with fixed routing and joint routing.

VI. PUBLICATION HISTORY

We give an overview of the publication history of TSN scheduling. First, we highlight seminal works from the literature. Then, we analyze the development of the field with respect to the number of published papers per year.

A. SEMINAL WORKS

Early works about per-flow scheduling in Ethernet networks were presented by Steiner [7] and Schweissguth et al. [8]. While these works are not specifically for TSN and abstract on the details of the real-time enhancement for Ethernet. they influenced many later works presented in this survey. The first works specifically about scheduling in TSN were presented in 2016. Dürr et al. [28] presented an ILP for no-wait scheduling and identified the problem of guard bands consuming bandwidth. Craciunas et al. [29] adapted the work of Steiner [7] for TSN. They introduced isolation constraints and incremental scheduling to the domain of TSN. Gavrilut et al. [73] is the first work which features joint routing and reliability considerations. Raagard et al. [76] introduced reconfiguration of schedules to TSN scheduling. Oliver et al. [31] proposed a scheduling approach with limits the number of used GCL entries by computing them in a joint approach with transmission offsets. All earlier works computed GCLs by a post-processing after scheduling.

B. PUBLISHED PAPERS

Figure 19 shows the number of papers about TSN scheduling per year. The first papers about scheduling in TSN were published in 2016. The general trend is that the field grows almost monotonously from one year to the next, with only one exception in 2019. We observe a significant increase in published works since the year 2020. Given the fast growth of the last 2 years, we expect even more research works about TSN scheduling in the future.

VII. FUTURE WORKS

In this section, we discuss the results of the literature study. First, we suggest improvements for future research works. Then, we highlight open problems not handled sufficiently so far.

A. SUGGESTIONS FOR IMPROVEMENT

The surveyed literature features many high quality research works. However, there is room for improvement in the presentation of some of these works. We suggest improvements in the hope that the overall quality of the TSN scheduling literature can be improved even more in the future. First, we discuss shortcomings and improvements in the presentation of evaluation methodologies. Then, we suggest the use of the technical terminology used in Ethernet bridging.

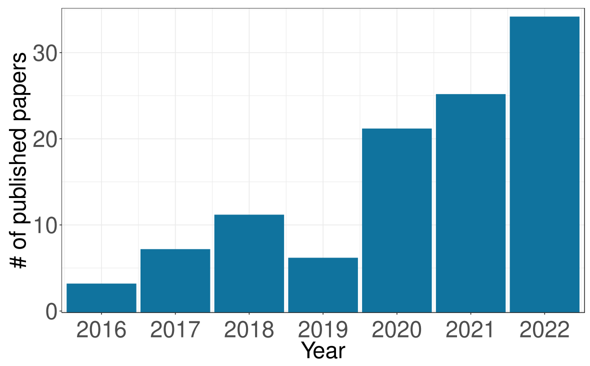


FIGURE 19. **Number of published research works per year about TSN scheduling. Only works published before March 2023 are counted.**

1) EVALUATION METHODOLOGIES

The scalability of the proposed solutions from the literature was extensively evaluated. Unfortunately, the impact of possible additional features and changes in parameters to the various objective functions was mostly ignored so far. Although scalability is an important property of a scheduling algorithm, it would be interesting to see more evaluations regarding solution quality. Most test cases in the literature are synthetically constructed, both network topology and streams. Even though it is hard to obtain test cases from the industry, let alone publish them, we would like to see more evaluations with realistic instances. It is not clear whether the proposed algorithms are suitable for large industry-scale instances or how they look like. It is also extremely difficult to compare the results of different research works as there is no public set of test cases for benchmarks. Consequently, there is little research work available about which algorithm should be used in which setting.

Unfortunately, it is also hard to assess the significance of evaluation results in some papers for two reasons. First, the instances solved are not sufficiently described. At least the topology and a description of assumed delays, e.g., processing and propagation delay, should be contained in the description of the network. Important properties of streams like deadlines or periods are often missing. Second, some evaluations report results for individual problem instances and are thus more of anecdotal character. There are easy and hard instances for every algorithm. Comparing multiple approaches on the same selected instances can be useful, but this may have the taste of picking specific instances in support of some conclusion. Instead of reporting results for individual instances, average results for multiple instances with the same evaluation setting should be reported. However, we acknowledge that evaluations for specific scenarios can be of interest. Especially readers from the industry may enjoy a rigorous report about a specific running system. We only want to emphasize that strong conclusions should be backed up by strong evidence and extensive evaluation. Another property covered by many evaluations is the schedulability of the respective proposed approach. These evaluations treat instances as infeasible when no schedule was found before some timeout. Thus, comparing the schedulability of two scheduling approaches which support different features is biased, as timeouts do not prove infeasibility. This may lead to wrong conclusions in favor of some algorithm or model, although schedulability is actually equal.

2) TERMINOLOGY

Many works surveyed in this paper use a vocabulary loosely related to Ethernet bridging. However, the standards and other relevant literature use a specific technical terminology. We suggest that the scheduling community adopts this jargon. Readers from adjacent research domains or who have prior knowledge in Ethernet bridging can benefit from a consistent vocabulary. The word *stream* is used in several standards, e.g., [11], [12], and [147]. Therefore, we suggest to use *stream* instead of *flow*. Network devices which send or receive data streams are denoted as *end stations* in the original bridging standard IEEE 802.1D [148]. The source end station of a stream is denoted as *talker*, while the destination end station is denoted as *listener*. Layer 2 switching devices are denoted as *bridges* instead of *switches* in IEEE 802.1D [148] and in the names of many other standards, e.g., in [9] and [14]. Thus, we suggest to use these terms when describing network topologies.

Frames are the units of data transmission, as TSN is a layer 2 technology, while packets are the units of data transmissions in layer 3 technologies (cf. [149]). Although the meaning of the term *packet* is clear in the context of scheduling, it is technically wrong. Routing is the process of path computation on layer 3. Therefore, the term *path selection* is more appropriate in TSN. However, we note that we used the term *routing* in this survey several times. The reason for this is to ensure consistency with the reviewed literature which solves the so-called joint routing problem.

B. OPEN PROBLEMS

The available literature is comprehensive with regard to solution approaches to the unmodified scheduling problem in TSN. However, there is still a wide field of relevant aspects which are not yet understood.

1) IMPACT OF GUARD BANDS AND GCL ENTRIES

To the best of our knowledge, the impact of guard bands on bandwidth available to lower-priority traffic was not evaluated in the literature. Likewise, the impact of available GCL entries on available bandwidth for lower-priority traffic is not investigated in detail. The evaluations so far suggest that AVB streams benefit from schedules with many holes between TT streams with regard to tardiness. However, such schedules may need more gate closings and thus guard bands, which reduces the available bandwidth. It is not clear how AVB and BE traffic can be simultaneously integrated in a unified approach for the scheduling of TT streams.

2) ROUTING AND MULTICAST

The joint routing and scheduling problem was explored in detail in the literature. All research works about this topic agree that schedulability benefits from joint routing and scheduling. However, solving the joint routing problem is significantly harder compared to scheduling with a given routing. Unfortunately, there is currently no exact and scalable approach known for joint routing. Additionally, it is not understood which properties a routing should have to benefit schedule synthesis and quality. TSN supports multicast streams which are relevant in real use cases. Some of the algorithms presented in research works covered by this survey can handle multicast streams. However, the literature lacks evaluations and insights about the appropriate integration of multicast streams in a schedule.

3) ONLINE RECONFIGURATION

There is also little work about online schedule reconfiguration, though it is important for operation. In some scenarios, e.g., automotive networks, insertion and removal of streams at execution time of the schedule can be important. So far it is not explored exhaustively what properties a schedule should have such that reconfiguration can by computed efficiently. However, there are preliminary works about this topic [79], [116]. Instead of rescheduling all streams, many works about reconfiguration try to resolve conflicts by assigning streams to other traffic classes [76] or paths [136], [138]. Other ways to resolve scheduling conflicts instead of rescheduling all streams may be needed in the future. Unfortunately, there are no reconfiguration algorithms for most of the problem extensions from Section III-E.

4) QUEUING AND HANDLING OF NON-DETERMINISM

An important open problem in TSN is sufficient integration of queuing. Almost all research works use isolation constraints from [29], i.e., they do not allow frames of different streams to reside in the same queue at the same time. However, this is not a requirement of the TAS. Some approaches even separate streams by assigning them to different egress queues during scheduling. The rational of this is to reduce the impact of non-determinism like frame loss. Other attempts to reduce the influence of such causes of non-determinism are not yet explored. The benefits of unrestricted queuing regarding schedulability or solution quality has not yet been evaluated.

Real hardware bridges are subject to non-determinism. There is jitter in processing delays, and clocks are not exactly synchronized in reality. Additionally, frames that are scheduled to arrive approximately at the same time at two ingress ports of the same bridge may cause race conditions, i.e., processing order is not deterministic. All research works covered by this survey assume bridges are perfectly deterministic. Thus, the literature lacks handling of such causes of non-determinism.

5) FILTERING AND POLICING

Per-Stream Filtering and Policing (PSFP) is a standard defined in IEEE 802.1Qci [147] for filtering and policing in TSN. Currently, there are no devices available implementing PSFP. However, filtering and policing could be used to prevent violations of schedules through unexpected packets. Packets not scheduled, delayed frames, and frames larger than expected can be filtered at execution time of a schedule. Thus, PSFP requires configuration of filtering entries that need to be derived from the schedule. A joint approach may be needed as PSFP imposes additional restrictions, e.g., the number of available filtering entries will be limited in bridges.

6) SECURITY ASPECTS

The security of real-time Ethernet networks was mostly ignored so far. All considerations for reliability and safety in TSN assume that no malicious party is involved in the communication. The standards do not cover countermeasures against replay or impersonation attacks. This may be a problem for highly vulnerable use cases of TSN, e.g., factory automation and in-vehicle networks. Future scheduling algorithms may integrate security considerations. For instance, key exchange and management result in additional streams that must be protected from other traffic. However, source authentication and integrity may also be implemented by future TSN standards or application layer protocols. Other security problems may be countered with PSFP, e.g., jamming and Denial-of-Service attacks by malicious end stations.

7) LEGACY DEVICES

Traditional Ethernet is a widespread layer 2 technology for industrial applications. Such applications are typically designed to be used for years or even decades. Thus, integrating legacy devices which are not capable of traffic scheduling or time synchronization will be a major problem in the next years for the deployment of TSN. Future scheduling algorithms may help to integrate such devices. For instance, scheduling algorithms can reserve bandwidth for the communication of these devices. However, new devices, e.g., gateways or proxies, may be needed to fully support the coexistence of legacy devices and scheduled traffic.

8) USE OF TSN MECHANISMS

TSN is not limited to scheduled traffic and the TAS. Other traffic classes may have real-time requirements, but cannot be scheduled as the respective streams are not periodic. Different traffic classes may have different sets of real-time requirements, e.g., demanding bounded jitter instead of bounded latency. TSN features more mechanisms which may be applied to fulfill these requirements, such as Asynchronous Traffic Shaping [150] or Cyclic Queuing and Forwarding [18]. A major open problem in TSN is the coexistence of multiple mechanisms and the assignment of streams to them. Input may be a set of streams or traffic rates with their descriptors and real-time requirements, and output is their assignment to appropriate TSN mechanisms together with the complete network configuration. This problem goes far beyond the TSN scheduling problem, but may impose additional constraints on the latter. Some requirements can only be fulfilled by scheduling the respective streams and computing GCLs for the TAS. Others may not even know the traffic streams in advance and can be implemented by the CBS or even simpler mechanisms. As even computing GCLs for the TAS is a challenging task for current state-of-the-art scheduling and optimization algorithms, such a comprehensive approach is currently unreachable. Hopefully, future works will move towards such long term goals and enable users to exploit the full potential of TSN.

9) UNDERSTANDING OF THE TSN PROBLEM

So far, scalability analyses have been conducted on special algorithms. However, they do not provide insights in what makes the TSN problem hard. This also pertains to all problem extensions like joint routing and multicast, reliability, robustness, BE or ABE traffic, etc. Moreover, properties of schedules such as tightness or average duration of open periods of the TAS have not yet been investigated. It would be helpful to understand the impact of problem extensions on the structure of schedules in an intuitive way. A better understanding of extensions and their impact on schedule structure may facilitate the development of heuristic algorithms that solve larger instances of the TSN problem with acceptable quality compared to exact approaches.

VIII. CONCLUSION

TSN is a set of standards to enable real-time transmission over switched Ethernet networks. IEEE 802.1Qbv [6] defines traffic scheduling combined with the Time-Aware Shaper (TAS), i.e., transmissions of periodic high-priority streams are scheduled such that packets hardly interfere and that ultra-low latency is achieved. Moreover, the TAS protects scheduled traffic against traffic from other traffic classes. This approach requires the configuration of transmission times for streams at the Talkers (source nodes) as well as the configuration of the TAS on the switches.

In this paper, we first gave an introduction to TSN with focus on traffic scheduling and the TAS. We defined the "TSN scheduling problem" and discussed common extensions such as scheduling with fixed or joint routing, various forms of queuing, support for reliability or lower-priority traffic, or respecting technical restrictions. Some of these extensions lead to optimization problems. We summarized frequently used scheduling and optimization methods to tackle these challenges. Then we reviewed a large body of literature about the TSN scheduling problem and classified it regarding the mentioned extensions. Subsequently, we analyzed and compared the works with respect to modelling assumptions, scheduling objectives, problem instances, and scalability, and pointed out advances. We tracked seminal works and identified popular publication venues for TSN scheduling. We discussed the area by suggesting improvements and pointing out open problems.

This survey serves researchers to identify the current state of the art and open problems in TSN scheduling. The many problem extensions suggest that the construction of an efficient scheduling or optimization algorithm which considers all relevant aspects is infeasible. We expect future work to provide a better understanding of the complexity of the TSN scheduling problem to cover more problem extensions while maintaining scalability.

ACKNOWLEDGMENT

The authors thank Manuel Eppler and Lukas Bechtel for valuable input and stimulating discussions.

The authors alone are responsible for the content of the paper.

REFERENCES

[1] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. ElBakoury, "Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G ULL research," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 1, pp. 88–145, 1st Quart., 2019.

[2] A. Minaeva and Z. Hanzálek, "Survey on periodic scheduling for time-triggered hard real-time systems," *ACM Comput. Surv.*, vol. 54, no. 1, pp. 1–32, Jan. 2022.

[3] Y. Seol, D. Hyeon, J. Min, M. Kim, and J. Paek, "Timely survey of time-sensitive networking: Past and future directions," *IEEE Access*, vol. 9, pp. 142506–142527, 2021.

[4] L. Deng, G. Xie, H. Liu, Y. Han, R. Li, and K. Li, "A survey of real-time Ethernet modeling and design methodologies: From AVB to TSN," *ACM Comput. Surv.*, vol. 55, no. 2, pp. 1–36, Feb. 2023.

[5] V. Gavriluţ, A. Pruski, and M. S. Berger, "Constructive or optimized: An overview of strategies to design networks for time-critical applications," *ACM Comput. Surv.*, vol. 55, no. 3, pp. 1–35, Mar. 2023.

[6] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 25: Enhancements for Scheduled Traffic*, IEEE Standard 802.1Qbv-2015, 2016, pp. 1–57.

[7] W. Steiner, "An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks," in *Proc. 31st IEEE Real-Time Syst. Symp.*, Nov. 2010, pp. 375–384.

[8] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegla, and G. Mühl, "ILP-based joint routing and scheduling for time-triggered networks," in *Proc. 25th Int. Conf. Real-Time Netw. Syst.*, Oct. 2017, pp. 8–17.

[9] *IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications*, IEEE Standard 802.1AS-2020, 2020, pp. 1–421.

[10] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Standard 1588-2019, 2020, pp. 1–499.

[11] *IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP)*, IEEE Standard 802.1Qat-2010, 2010, pp. 1–119.

[12] *IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams*, IEEE Standard 802.1Qav-2009, 2010, pp. C1–72.

[13] *IEEE Standard for Local and Metropolitan Area Networks–Audio Video Bridging (AVB) Systems*, IEEE Standard 802.1BA-2021, 2021, pp. 1–45.

[14] *IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks*, IEEE Standard 802.1Q-2018, 2018, pp. 1–1993.

[15] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 24: Path Control and Reservation*, IEEE Standard 802.1Qca-2015, 2016, pp. 1–120.

[16] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 26: Frame Preemption*, IEEE Standard 802.1Qbu-2016, 2016, pp. 1–52.

[17] *IEEE Standard for Local and Metropolitan Area Networks–Frame Replication and Elimination for Reliability*, IEEE Standard 802.1CB-2017, 2017, pp. 1–102.

[18] *IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks–Amendment 29: Cyclic Queuing and Forwarding*, Standard IEEE 802.1Qch-2017, 2017, pp. 1–30.

[19] G. Patti, L. L. Bello, and L. Leonardi, "Deadline-aware online scheduling of TSN flows for automotive applications," *IEEE Trans. Ind. Informat.*, vol. 19, no. 4, pp. 5774–5784, Apr. 2023.

[20] M. Kim, J. Min, D. Hyeon, and J. Paek, "TAS scheduling for real-time forwarding of emergency event traffic in TSN," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2020, pp. 1111–1113.

[21] J. Xue, G. Shou, Y. Liu, Y. Hu, and Z. Guo, "Time-aware traffic scheduling with virtual queues in time-sensitive networking," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, May 2021, pp. 604–607.

[22] Y. Zhang, Q. Xu, S. Wang, Y. Chen, L. Xu, and C. Chen, "Scalable no-wait scheduling with flow-aware model conversion in time-sensitive networking," in *Proc. IEEE Global Commun. Conf.*, Dec. 2022, pp. 413–418.

[23] Z. Cao, Q. Liu, D. Liu, and Y. Hu, "Enhanced system design and scheduling strategy for switches in time-sensitive networking," *IEEE Access*, vol. 9, pp. 42621–42634, 2021.

[24] Y. Zhang, J. Wu, M. Liu, and A. Tan, "TSN-based routing and scheduling scheme for industrial Internet of Things in underground mining," *Eng. Appl. Artif. Intell.*, vol. 115, Oct. 2022, Art. no. 105314.

[25] Y. Zhang, Q. Xu, L. Xu, C. Chen, and X. Guan, "Efficient flow scheduling for industrial time-sensitive networking: A divisibility theory-based method," *IEEE Trans. Ind. Informat.*, vol. 18, no. 12, pp. 9312–9323, Dec. 2022.

[26] W. Han, Y. Li, and C. Yin, "A traffic scheduling algorithm combined with ingress shaping in TSN," in *Proc. 14th Int. Conf. Wireless Commun. Signal Process. (WCSP)*, Nov. 2022, pp. 586–591.

[27] M. Wei and S. Yang, "A network scheduling method for convergence of industrial wireless network and TSN," in *Proc. 17th Int. Conf. Ubiquitous Inf. Manage. Commun. (IMCOM)*, Jan. 2023, pp. 1–6.

[28] F. Dürr and N. G. Nayak, "No-wait packet scheduling for IEEE time-sensitive networks (TSN)," in *Proc. 24th Int. Conf. Real-Time Netw. Syst.*, Oct. 2016, pp. 1–15.

[29] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling real-time communication in IEEE 802.1 Qbv time sensitive networks," in *Proc. 24th Int. Conf. Real-Time Netw. Syst.*, Oct. 2016, pp. 183–192.

[30] X. Jin, C. Xia, N. Guan, C. Xu, D. Li, Y. Yin, and P. Zeng, "Real-time scheduling of massive data in time sensitive networks with a limited number of schedule entries," *IEEE Access*, vol. 8, pp. 6751–6767, 2020.

[31] R. Serna Oliver, S. S. Craciunas, and W. Steiner, "IEEE 802.1 Qbv gate control list synthesis using array theory encoding," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2018, pp. 13–24.

[32] A. Alnajim, S. Salehi, and C. Shen, "Incremental path-selection and scheduling for time-sensitive networks," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2019, pp. 1–6.

[33] A. A. Syed, S. Ayaz, T. Leinmüller, and M. Chandra, "Fault-tolerant dynamic scheduling and routing for TSN based in-vehicle networks," in *Proc. IEEE Veh. Netw. Conf. (VNC)*, Nov. 2021, pp. 72–75.

[34] Cplex, IBM ILOG, "V12.1: Users manual for CPLEX," *Int. Bus. Mach. Corp.*, vol. 46, no. 53, p. 157, 2009.

[35] Gurobi Optimization, LLC. (2021). *Gurobi Optimizer Reference Manual*. [Online]. Available: https://www.gurobi.com

[36] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Theory Pract. Softw., Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2008, pp. 337–340.

[37] L. Perron and V. Furnon. (Jul. 2019). *OR-Tools*. Google. [Online]. Available: https://developers.google.com/optimization/

[38] E. Alpaydin, *Introduction to Machine Learning*. Cambridge, MA, USA: MIT Press, 2020.

[39] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[40] D. Hellmanns, A. Glavackij, J. Falk, R. Hummen, S. Kehrer, and F. Dürr, "Scaling TSN scheduling for factory automation networks," in *Proc. 16th IEEE Int. Conf. Factory Commun. Syst. (WFCS)*, Apr. 2020, pp. 1–8.

[41] W. Steiner, S. S. Craciunas, and R. S. Oliver, "Traffic planning for time-sensitive communication," *IEEE Commun. Standards Mag.*, vol. 2, no. 2, pp. 42–47, Jun. 2018.

[42] H.-J. Kim, M.-H. Choi, M.-H. Kim, and S. Lee, "Development of an Ethernet-based heuristic time-sensitive networking scheduling algorithm for real-time in-vehicle data transmission," *Electronics*, vol. 10, no. 2, p. 157, Jan. 2021.

[43] H. J. Kim, K. C. Lee, and S. Lee, "A genetic algorithm based scheduling method for automotive Ethernet," in *Proc. 47th Annu. Conf. IEEE Ind. Electron. Soc.*, Oct. 2021, pp. 1–5.

[44] H.-J. Kim, K.-C. Lee, M.-H. Kim, and S. Lee, "Optimal scheduling of time-sensitive networks for automotive Ethernet based on genetic algorithm," *Electronics*, vol. 11, no. 6, p. 926, Mar. 2022.

[45] F. Ansah, M. A. Abid, and H. de Meer, "Schedulability analysis and GCL computation for time-sensitive networks," in *Proc. IEEE 17th Int. Conf. Ind. Informat. (INDIN)*, vol. 1, Jul. 2019, pp. 926–932.

[46] H. Wang, Z. Zhao, and J. Wei, "Adaptive scheduling algorithm based on time aware shaper," in *Proc. 5th Int. Conf. Adv. Electron. Mater., Comput. Softw. Eng. (AEMCSE)*, Apr. 2022, pp. 555–562.

[47] J. Min, M. Oh, W. Kim, H. Seo, and J. Paek, "Evaluation of metaheuristic algorithms for TAS scheduling in time-sensitive networking," in *Proc. 13th Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2022, pp. 809–812.

[48] M. Vlk, K. Brejchová, Z. Hanzálek, and S. Tang, "Large-scale periodic scheduling in time-sensitive networks," *Comput. Oper. Res.*, vol. 137, Jan. 2022, Art. no. 105512.

[49] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.

[50] X. Wang, H. Yao, T. Mai, T. Nie, L. Zhu, and Y. Liu, "Deep reinforcement learning aided no-wait flow scheduling in time-sensitive networks," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2022, pp. 812–817.

[51] M. Vlk, Z. Hanzálek, K. Brejchová, S. Tang, S. Bhattacharjee, and S. Fu, "Enhancing schedulability and throughput of time-triggered traffic in IEEE 802.1 Qbv time-sensitive networks," *IEEE Trans. Commun.*, vol. 68, no. 11, pp. 7023–7038, Nov. 2020.

[52] N. Reusch, L. Zhao, S. S. Craciunas, and P. Pop, "Window-based schedule synthesis for industrial IEEE 802.1 Qbv TSN networks," in *Proc. 16th IEEE Int. Conf. Factory Commun. Syst. (WFCS)*, Apr. 2020, pp. 1–4.

[53] P.-J. Chaine, M. Boyer, C. Pagetti, and F. Wartel, "Egress-TT configurations for TSN networks," in *Proc. 30th Int. Conf. Real-Time Netw. Syst.*, Jun. 2022, p. 5869.

[54] D. Bujosa, M. Ashjaei, A. V. Papadopoulos, T. Nolte, and J. Proenza, "HERMES: Heuristic multi-queue scheduler for TSN time-triggered traffic with zero reception jitter capabilities," in *Proc. 30th Int. Conf. Real-Time Netw. Syst.*, Jun. 2022, p. 7080.

[55] N. Reusch, S. S. Craciunas, and P. Pop, "Dependability-aware routing and scheduling for time-sensitive networking," *IET Cyber-Phys. Syst., Theory Appl.*, vol. 7, no. 3, pp. 124–146, Sep. 2022.

[56] N. G. Nayak, "Scheduling & routing time-triggered traffic in time-sensitive networks," Ph.D. dissertation, Graduate School Excellence Adv. Manuf. Eng. (GSaME), Univ. Stuttgart, Stuttgart, Germany, Tech. Rep., 2018.

[57] P. Pop, M. L. Raagaard, S. S. Craciunas, and W. Steiner, "Design optimisation of cyber-physical distributed systems using IEEE time-sensitive networks," *IET Cyber-Phys. Syst., Theory Appl.*, vol. 1, no. 1, pp. 86–94, Dec. 2016.

[58] E. Li, F. He, L. Zhao, and X. Zhou, "A SDN-based traffic bandwidth allocation method for time sensitive networking in avionics," in *Proc. IEEE/AIAA 38th Digit. Avionics Syst. Conf. (DASC)*, Sep. 2019, pp. 1–7.

[59] A. C. T. dos Santos, B. Schneider, and V. Nigam, "TSNSCHED: Automated schedule generation for time sensitive networking," in *Proc. Formal Methods Comput. Aided Design*, 2019, pp. 69–77.

[60] A. Varga, *OMNeT++*. Berlin, Germany: Springer, 2010, pp. 35–59.

[61] A. C. T. D. Santos, "TSNsched: Automated schedule generation for time sensitive networking," Ph.D. dissertation, Centro de Informática, Universidade Federal da Paraíba, Paraíba, Brazil, 2020.

[62] B. Houtan, M. Ashjaei, M. Daneshtalab, M. Sjödin, and S. Mubeen, "Synthesising schedules to improve QoS of best-effort traffic in TSN networks," in *Proc. 29th Int. Conf. Real-Time Netw. Syst.*, Apr. 2021, pp. 68–77.

[63] W. Zhou and Z. Li, "Implementation and evaluation of SMT-based real-time communication scheduling for IEEE 802.1 Qbv in next-generation in-vehicle network," in *Proc. 2nd Int. Conf. Inf. Technol. Comput. Appl. (ITCA)*, Dec. 2020, pp. 457–461.

[64] M. Barzegaran, N. Reusch, L. Zhao, S. S. Craciunas, and P. Pop, "Real-time traffic guarantees in heterogeneous time-sensitive networks," in *Proc. 30th Int. Conf. Real-Time Netw. Syst.*, Jun. 2022, p. 4657.

[65] J. Pei, Y. Hu, L. Tian, M. Li, and Z. Li, "A hybrid traffic scheduling strategy for time-sensitive networking," *Electronics*, vol. 11, no. 22, p. 3762, Nov. 2022.

[66] X. Yao, Z. Gan, Y. Chen, L. Guo, and W. Wang, "Hybrid flow scheduling with additional simple compensation mechanisms in time-sensitive networks," in *Proc. IEEE 6th Adv. Inf. Technol., Electron. Autom. Control Conf. (IAEAC)*, Oct. 2022, pp. 1315–1320.

[67] S. Wang, Q. Xu, Y. Zhang, L. Xu, and C. Chen, "Hybrid traffic scheduling based on adaptive time slot slicing in time-sensitive networking," in *Proc. IEEE Int. Conf. Ind. Technol. (ICIT)*, Aug. 2022, pp. 1–7.

[68] Z. Huang, H. Zhu, H. Zhang, and T. Huang, "A scalable heuristic time-sensitive traffic scheduling algorithm for in-vehicle network," in *Proc. 5th Int. Conf. Hot Inf.-Centric Netw. (HotICN)*, Nov. 2022, pp. 111–118.

[69] S. S. Craciunas and R. S. Oliver, "Out-of-sync schedule robustness for time-sensitive networks," in *Proc. 17th IEEE Int. Conf. Factory Commun. Syst. (WFCS)*, Jun. 2021, pp. 75–82.

[70] Z. Feng, M. Cai, and Q. Deng, "An efficient pro-active fault-tolerance scheduling of IEEE 802.1 Qbv time-sensitive network," *IEEE Internet Things J.*, vol. 9, no. 16, pp. 14501–14510, Aug. 2022.

[71] A. A. Atallah, G. B. Hamad, and O. A. Mohamed, "Routing and scheduling of time-triggered traffic in time-sensitive networks," *IEEE Trans. Ind. Informat.*, vol. 16, no. 7, pp. 4525–4534, Jul. 2020.

[72] N. Reusch, P. Pop, and S. S. Craciunas, "Work-in-progress: Safe and secure configuration synthesis for TSN using constraint programming," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2020, pp. 387–390.

[73] V. Gavrilut, B. Zarrin, P. Pop, and S. Samii, "Fault-tolerant topology and routing synthesis for IEEE time-sensitive networking," in *Proc. 25th Int. Conf. Real-Time Netw. Syst.*, Oct. 2017, pp. 267–276.

[74] Z. Feng, Q. Deng, M. Cai, and J. Li, "Efficient reservation-based fault-tolerant scheduling for IEEE 802.1 Qbv time-sensitive networking," *J. Syst. Archit.*, vol. 123, Feb. 2022, Art. no. 102381.

[75] R. Dobrin, N. Desai, and S. Punnekkat, "On fault-tolerant scheduling of time sensitive networks," in *Proc. Int. Workshop Secur. Dependability Crit. Embedded Real-Time Syst.*, vol. 73, 2019, pp. 5:1–5:13.

[76] M. L. Raagaard, P. Pop, M. Gutiérrez, and W. Steiner, "Runtime reconfiguration of time-sensitive networking (TSN) schedules for fog computing," in *Proc. IEEE Fog World Congr. (FWC)*, Oct. 2017, pp. 1–6.

[77] Z. Pang, X. Huang, Z. Li, S. Zhang, Y. Xu, H. Wan, and X. Zhao, "Flow scheduling for conflict-free network updates in time-sensitive software-defined networks," *IEEE Trans. Ind. Informat.*, vol. 17, no. 3, pp. 1668–1678, Mar. 2021.

[78] Y. Wang, F. Wang, W. Wang, X. Tan, J. Wen, Y. Wang, and P. Lin, "Design and implementation of traffic scheduling algorithm for time-sensitive network," in *Proc. IEEE Int. Symp. Broadband Multimedia Syst. Broadcast. (BMSB)*, Jun. 2022, pp. 1–6.

[79] C. Gärtner, A. Rizk, B. Koldehofe, R. Guillaume, R. Kundel, and R. Steinmetz, "On the incremental reconfiguration of time-sensitive networks at runtime," in *Proc. IFIP Netw. Conf.*, Jun. 2022, pp. 1–9.

[80] C. Gärtner, A. Rizk, B. Koldehofe, R. Guillaume, R. Kundel, and R. Steinmetz, "Fast incremental reconfiguration of dynamic time-sensitive networks at runtime," *Comput. Netw.*, vol. 224, Apr. 2023, Art. no. 109606.

[81] Q. Li, D. Li, X. Jin, Q. Wang, and P. Zeng, "A simple and efficient time-sensitive networking traffic scheduling method for industrial scenarios," *Electronics*, vol. 9, no. 12, p. 2131, Dec. 2020.

[82] J. Dai, Z. Wang, and L. Zhong, "Research on gating scheduling of time sensitive network based on constraint strategy," *J. Phys., Conf. Ser.*, vol. 1920, no. 1, May 2021, Art. no. 012089.

[83] T. Feng and H. Yang, "SMT-based task- and network-level static schedule for time sensitive network," in *Proc. Int. Conf. Commun., Inf. Syst. Comput. Eng. (CISCE)*, May 2021, pp. 764–770.

[84] M. Barzegaran, B. Zarrin, and P. Pop, "Quality-of-control-aware scheduling of communication in TSN-based fog computing platforms using constraint programming," in *Proc. Workshop Fog Comput. IoT*, 2020, pp. 3:1–3:9.

[85] M. Barzegaran and P. Pop, "Communication scheduling for control performance in TSN-based fog computing platforms," *IEEE Access*, vol. 9, pp. 50782–50797, 2021.

[86] M. Barzegaran, "Configuration optimization of fog computing platforms for control applications," Ph.D. dissertation, Dept. Appl. Math. Comput. Sci., Tech. Univ. Denmark, Lyngby, Denmark, 2021.

[87] S. D. McLean, E. A. Juul Hansen, P. Pop, and S. S. Craciunas, "Configuring ADAS platforms for automotive applications using metaheuristics," *Frontiers Robot. AI*, vol. 8, pp. 1–15, Jan. 2022.

[88] A. Arestova, W. Baron, K. J. Hielscher, and R. German, "ITANS: Incremental task and network scheduling for time-sensitive networks," *IEEE Open J. Intell. Transp. Syst.*, vol. 3, pp. 369–387, 2022.

[89] X. Jin, C. Xia, N. Guan, and P. Zeng, "Joint algorithm of message fragmentation and no-wait scheduling for time-sensitive networks," *IEEE/CAA J. Autom. Sinica*, vol. 8, no. 2, pp. 478–490, Feb. 2021.

[90] T. Park, S. Samii, and K. G. Shin, "Design optimization of frame preemption in real-time switched Ethernet," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 420–425.

[91] D. Ginthör, R. Guillaume, J. von Hoyningen-Huene, M. Schüngel, and H. D. Schotten, "End-to-end optimized joint scheduling of converged wireless and wired time-sensitive networks," in *Proc. 25th IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, vol. 1, Sep. 2020, pp. 222–229.

[92] J. Lin, W. Li, X. Feng, S. Zhan, J. Feng, J. Cheng, T. Wang, Q. Li, Y. Wang, F. Li, and B. Tang, "Rethinking the use of network cycle in time-sensitive networking (TSN) flow scheduling," in *Proc. IEEE/ACM 30th Int. Symp. Quality Service (IWQoS)*, Jun. 2022, pp. 1–11.

[93] M. H. Farzaneh, S. Kugele, and A. Knoll, "A graphical modeling tool supporting automated schedule synthesis for time-sensitive networking," in *Proc. 22nd IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2017, pp. 1–8.

[94] J. Tu, Q. Xu, L. Xu, and C. Chen, "SSL-SP: A semi-supervised-learning-based stream partitioning method for scale iterated scheduling in time-sensitive networks," in *Proc. 22nd IEEE Int. Conf. Ind. Technol. (ICIT)*, vol. 1, Mar. 2021, pp. 1182–1187.

[95] R. Mahfouzi, A. Aminifar, S. Samii, A. Rezine, P. Eles, and Z. Peng, "Stability-aware integrated routing and scheduling for control applications in Ethernet networks," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 682–687.

[96] J. Falk, F. Dürr, and K. Rothermel, "Exploring practical limitations of joint routing and scheduling for TSN with ILP," in *Proc. IEEE 24th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2018, pp. 136–146.

[97] H. Nie, S. Li, and Y. Liu, "An enhanced routing and scheduling mechanism for time-triggered traffic with large period differences in time-sensitive networking," *Appl. Sci.*, vol. 12, no. 9, p. 4448, Apr. 2022.

[98] Y. Huang, S. Wang, T. Huang, B. Wu, Y. Wu, and Y. Liu, "Online routing and scheduling for time-sensitive networks," in *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2021, pp. 272–281.

[99] L. Xu, Q. Xu, J. Tu, J. Zhang, Y. Zhang, C. Chen, and X. Guan, "Learning-based scalable scheduling and routing co-design with stream similarity partitioning for time-sensitive networking," *IEEE Internet Things J.*, vol. 9, no. 15, pp. 13353–13363, Aug. 2022.

[100] M. Pahlevan, N. Tabassam, and R. Obermaisser, "Heuristic list scheduler for time triggered traffic in time sensitive networks," *ACM SIGBED Rev.*, vol. 16, no. 1, pp. 15–20, Feb. 2019.

[101] L. Xu, Q. Xu, Y. Zhang, J. Zhang, and C. Chen, "Co-design approach of scheduling and routing in time sensitive networking," in *Proc. IEEE Conf. Ind. Cyberphys. Syst. (ICPS)*, vol. 1, Jun. 2020, pp. 111–116.

[102] F. Smirnov, M. Glaß, F. Reimann, and J. Teich, "Optimizing message routing and scheduling in automotive mixed-criticality time-triggered networks," in *Proc. 54th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2017, pp. 1–6.

[103] A. Arestova, K.-S. J. Hielscher, and R. German, "Design of a hybrid genetic algorithm for time-sensitive networking," in *Proc. Int. Conf. Meas., Modeling Eval. Comput. Syst.*, 2020, pp. 99–117.

[104] M. Pahlevan and R. Obermaisser, "Genetic algorithm for scheduling time-triggered traffic in time-sensitive networks," in *Proc. IEEE 23rd Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, vol. 1, Sep. 2018, pp. 337–344.

[105] M. Nawaz, E. E. Enscore, and I. Ham, "A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem," *Omega*, vol. 11, no. 1, pp. 91–95, Jan. 1983.

[106] A. M. Kentis, M. S. Berger, and J. Soler, "Effects of port congestion in the gate control list scheduling of time sensitive networks," in *Proc. 8th Int. Conf. Netw. Future (NOF)*, Nov. 2017, pp. 138–140.

[107] Y. Wang, J. Chen, W. Ning, H. Yu, S. Lin, Z. Wang, G. Pang, and C. Chen, "A time-sensitive network scheduling algorithm based on improved ant colony optimization," *Alexandria Eng. J.*, vol. 60, no. 1, pp. 107–114, Feb. 2021.

[108] J. Falk, F. Dürr, and K. Rothermel, "Time-triggered traffic planning for data networks with conflict graphs," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2020, pp. 124–136.

[109] M. Vlk, Z. Hanzálek, and S. Tang, "Constraint programming approaches to joint routing and scheduling in time-sensitive networks," *Comput. Ind. Eng.*, vol. 157, Jul. 2021, Art. no. 107317.

[110] B. Caddell, "Joint routing and scheduling with SMT," B.Sc. thesis, Inst. Parallel Distrib. Syst., Univ. Stuttgart, Stuttgart, Germany, 2018.

[111] X. He, X. Zhuge, F. Dang, W. Xu, and Z. Yang, "DeepScheduler: Enabling flow-aware scheduling in time-sensitive networking," in *Proc. IEEE INFOCOM*, Mar. 2023, pp. 1–4.

[112] F. Pozo, G. Rodriguez-Navas, and H. Hansson, "Schedule reparability: Enhancing time-triggered network recovery upon link failures," in *Proc. IEEE 24th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2018, pp. 147–156.

[113] A. A. Atallah, G. B. Hamad, and O. A. Mohamed, "Fault-resilient topology planning and traffic configuration for IEEE 802.1 Qbv TSN networks," in *Proc. IEEE 24th Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Jul. 2018, pp. 151–156.

[114] Y. Zhou, S. Samii, P. Eles, and Z. Peng, "Reliability-aware scheduling and routing for messages in time-sensitive networking," *ACM Trans. Embedded Comput. Syst.*, vol. 20, no. 5, pp. 1–24, Sep. 2021.

[115] A. A. Syed, S. Ayaz, T. Leinmüller, and M. Chandra, "Fault-tolerant static scheduling and routing for in-vehicle networks," in *Proc. 32nd Int. Telecommun. Netw. Appl. Conf. (ITNAC)*, Nov. 2022, pp. 273–279.

[116] A. A. Syed, S. Ayaz, T. Leinmüller, and M. Chandra, "MIP-based joint scheduling and routing with load balancing for TSN based in-vehicle networks," in *Proc. IEEE Veh. Netw. Conf. (VNC)*, Dec. 2020, pp. 1–7.

[117] A. A. Syed, S. Ayaz, T. Leinmüller, and M. Chandra, "Network coding based fault-tolerant dynamic scheduling and routing for in-vehicle networks," *J. Netw. Syst. Manage.*, vol. 31, no. 1, p. 27, Jan. 2023.

[118] Y. Zhou, S. Samii, P. Eles, and Z. Peng, "ASIL-decomposition based routing and scheduling in safety-critical time-sensitive networking," in *Proc. IEEE 27th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, May 2021, pp. 184–195.

[119] *Road Vehicles: Functional Safety*, Standard ISO 26262, 2018.

[120] Y. Zhou, "Synthesis of safety-critical real-time systems," Ph.D. dissertation, Dept. Comput. Inf. Sci., Linköping Univ., Linköping, Sweden, 2022.

[121] Y. Zhou, S. Samii, P. Eles, and Z. Peng, "Time-triggered scheduling for time-sensitive networking with preemption," in *Proc. 27th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2022, pp. 262–267.

[122] H. Li, H. Cheng, and L. Yang, "Reliable routing and scheduling in time-sensitive networks," in *Proc. 17th Int. Conf. Mobility, Sens. Netw. (MSN)*, Dec. 2021, pp. 806–811.

[123] V. Gavrilut, L. Zhao, M. L. Raagaard, and P. Pop, "AVB-aware routing and scheduling of time-triggered traffic for TSN," *IEEE Access*, vol. 6, pp. 75229–75243, 2018.

[124] S. M. Laursen, P. Pop, and W. Steiner, "Routing optimization of AVB streams in TSN networks," *SIGBED Rev.*, vol. 13, no. 4, p. 4348, 2016.

[125] V. Gavrilut and P. Pop, "Scheduling in time sensitive networks (TSN) for mixed-criticality industrial applications," in *Proc. 14th IEEE Int. Workshop Factory Commun. Syst. (WFCS)*, Jun. 2018, pp. 1–4.

[126] M. L. Raagaard and P. Pop, "Optimization algorithms for the scheduling of IEEE 802.1 time-sensitive networking (TSN)," Tech. Univ. Denmark, Lyngby, Denmark, Tech. Rep. 1, 2017.

[127] C. Chuang, T. Yu, C. Lin, A. Pang, and T. Hsieh, "Online stream-aware routing for TSN-based industrial control systems," in *Proc. 25th IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, vol. 1, Sep. 2020, pp. 254–261.

[128] V. Gavriluţ and P. Pop, "Traffic-type assignment for TSN-based mixed-criticality cyber-physical systems," *ACM Trans. Cyber-Phys. Syst.*, vol. 4, no. 2, pp. 1–27, Apr. 2020.

[129] A. Berisa, L. Zhao, S. S. Craciunas, M. Ashjaei, S. Mubeen, M. Daneshtalab, and M. Sjödin, "AVB-aware routing and scheduling for critical traffic in time-sensitive networks with preemption," in *Proc. 30th Int. Conf. Real-Time Netw. Syst.*, Jun. 2022, pp. 207–218.

[130] Y. Li, J. Jiang, and S. H. Hong, "Joint traffic routing and scheduling algorithm eliminating the nondeterministic interruption for TSN networks used in IIoT," *IEEE Internet Things J.*, vol. 9, no. 19, pp. 18663–18680, Oct. 2022.

[131] L. Yang, Y. Wei, F. R. Yu, and Z. Han, "Joint routing and scheduling optimization in time-sensitive networks using graph-convolutional-network-based deep reinforcement learning," *IEEE Internet Things J.*, vol. 9, no. 23, pp. 23981–23994, Dec. 2022.

[132] E. Schweissguth, D. Timmermann, H. Parzyjegla, P. Danielis, and G. Mühl, "ILP-based routing and scheduling of multicast realtime traffic in time-sensitive networks," in *Proc. IEEE 26th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2020, pp. 1–11.

[133] C. Li, C. Zhang, W. Zheng, X. Wen, Z. Lu, and J. Zhao, "Joint routing and scheduling for dynamic applications in multicast time-sensitive networks," in *Proc. IEEE Int. Conf. Commun. Workshops*, Jun. 2021, pp. 1–6.

[134] Q. Yu and M. Gu, "Adaptive group routing and scheduling in multicast time-sensitive networks," *IEEE Access*, vol. 8, pp. 37855–37865, 2020.

[135] A. A. Syed, S. Ayaz, T. Leinmüller, and M. Chandra, "Dynamic scheduling and routing for TSN based in-vehicle networks," in *Proc. IEEE Int. Conf. Commun. Workshops*, Jun. 2021, pp. 1–6.

[136] Q. Yu, H. Wan, X. Zhao, Y. Gao, and M. Gu, "Online scheduling for dynamic VM migration in multicast time-sensitive networks," *IEEE Trans. Ind. Informat.*, vol. 16, no. 6, pp. 3778–3788, Jun. 2020.

[137] L. Kou, G. Markowsky, and L. Berman, "A fast algorithm for Steiner trees," *Acta Inf.*, vol. 15, no. 2, pp. 141–145, 1981.

[138] J. Li, H. Xiong, Q. Li, F. Xiong, and J. Feng, "Run-time reconfiguration strategy and implementation of time-triggered networks," *Electronics*, vol. 11, no. 9, p. 1477, May 2022.

[139] N. G. Nayak, F. Dürr, and K. Rothermel, "Incremental flow scheduling and routing in time-sensitive software-defined networks," *IEEE Trans. Ind. Informat.*, vol. 14, no. 5, pp. 2066–2075, May 2018.

[140] R. Mahfouzi, A. Aminifar, S. Samii, A. Rezine, P. Eles, and Z. Peng, "Breaking silos to guarantee control stability with communication over Ethernet TSN," *IEEE Des. Test. IEEE Des. Test. Comput.*, vol. 38, no. 5, pp. 48–56, Oct. 2021.

[141] Y. Zheng, S. Wang, S. Yin, B. Wu, and Y. Liu, "Mix-flow scheduling for concurrent multipath transmission in time-sensitive networking," in *Proc. IEEE Int. Conf. Commun. Workshops (ICC Workshops)*, Jun. 2021, pp. 1–6.

[142] D. Yang, K. Gong, J. Ren, W. Zhang, W. Wu, and H. Zhang, "TC-flow: Chain flow scheduling for advanced industrial applications in time-sensitive networks," *IEEE Netw.*, vol. 36, no. 2, pp. 16–24, Mar. 2022.

[143] K. Gong, D. Yang, W. Zhang, and J. Ren, "An efficient scheduling approach for multi-level industrial chain flows in time-sensitive networking," *Comput. Netw.*, vol. 221, Feb. 2023, Art. no. 109516.

[144] D. Hellmanns, L. Haug, M. Hildebrand, F. Dürr, S. Kehrer, and R. Hummen, "How to optimize joint routing and scheduling models for TSN using integer linear programming," in *Proc. 29th Int. Conf. Real-Time Netw. Syst.*, Apr. 2021, pp. 100–111.

[145] S. Bhattacharjee, K. Alexandris, E. Hansen, P. Pop, and T. Bauschert, "Latency-aware function placement, routing, and scheduling in TSN-based industrial networks," in *Proc. IEEE Int. Conf. Commun.*, May 2022, pp. 4248–4254.

[146] A. Perrig, R. Canetti, D. Song, and J. D. Tygar, "Efficient and secure source authentication for multicast," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, vol. 1, 2001, pp. 35–46.

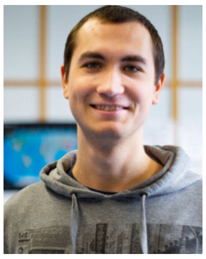
[147] *IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks–Amendment 28: Per-Stream Filtering and Policing*, IEEE Standard 802.1Qci-2017, 2017, pp. 1–65.

[148] *Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges*, IEEE Standard 802.1D-1990, 1991, pp. 1–176.

[149] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*. London, U.K.: Pearson, 2016.

[150] *IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks—Amendment 34: Asynchronous Traffic Shaping*, IEEE Standard 802.1Qcr-2020, 2020, pp. 1–151.

THOMAS STÜBER received the master's degree, in 2018. He is currently pursuing the Ph.D. degree with the Chair of Communication Networks of Prof. Dr. Habil. Michael Menth, University of Tübingen, Germany. He became part of the Communication Networks Research Group. His research interests include time-sensitive networking (TSN), scheduling, performance evaluation, and operations research.



LUKAS OSSWALD received the master's degree, in 2020. He is currently pursuing the Ph.D. degree with the Chair of Communication Networks of Prof. Dr. Habil. Michael Menth, University of Tübingen, Germany. He became part of the Communication Networks Research Group. His research interests include time-sensitive networking (TSN), admission control, and network configuration.

STEFFEN LINDNER received the master's degree, in 2019. He is currently pursuing the Ph.D. degree with the Chair of Communication Networks of Prof. Dr. Habil. Michael Menth, University of Tübingen, Germany. He became part of the Communication Networks Research Group. His research interests include software-defined networking, P4, and congestion management.

MICHAEL MENTH (Senior Member, IEEE) received the Diploma degree from The University of Texas at Austin, Austin, TX, USA, in 1998, the Ph.D. degree from Ulm University, Germany, in 2004, and the Habilitation degree from the University of Würzburg, Germany, in 2010. He has been the Chair Holder of Communication Networks, since 2010. He is currently a Professor with the Department of Computer Science, University of Tübingen, Germany. His special interests are performance analysis and optimization of communication networks, resilience and routing issues, as well as resource and congestion management. His recent research interests include network softwarization, in particular P4-based data plane programming, time-sensitive networking (TSN), the Internet of Things, and internet protocols. He contributes to standardization bodies, mainly to the IETF.