A Virtual Testbed Orchestration (VITO) System for Education, Performance Evaluation, and Network Function Virtualization

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

vorgelegt von
Andreas Stockmayer
aus Freiburg im Breisgau

Tübingen
2023

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

Tag der mündlichen Qualifikation:	09.07.2024
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter/-in:	Prof. Dr. Michael Menth
2. Berichterstatter/-in:	Prof. Dr. Andreas Zell

Kurzfassung

Der Paradigmenwechsel innerhalb der Informatikbranche, weg von dedizierter, spezialisierter Hardware, hin zu virtualisierten Systemen auf günstiger generischer Hardware bietet auch im Netzwerkbereich grosses Potential. Diese Arbeit fokussiert sich darauf, wie diese neuen Technologien im Netzwerkumfeld verwendet werden können, um sowohl die Arbeit an Universitäten als auch in Datacenters zu vereinfachen und die Kosten zu optimieren. Aus dem universitären Kontext wird in dieser Arbeit ein System vorgestellt, welches die Ausbildung von Studierenden erleichtert. Für Forschende wird ein Framework vorgestellt, mit dem die Evaluierung neuer Protokolle und Technologien in virtualisierten Umgebungen ermöglicht wird. Das letzte Projekt beschäftigt sich mit einem System, um Datacenter-Betreibern den Zugang zu moderner Netzwerkvirtualisierungstechnik zu ermöglichen, ohne dabei auf spezialisierte Hardware zurückgreifen zu müssen. Alle in dieser Arbeit beschriebenen Systeme richten sich nicht nur an Netzwerkspezialisten, sondern generell an Administratoren mit grundlegenden Netzwerkkentnissen.

Abstract

In recent years, the computer science world has seen a big shift from dedicated, highly specialised hardware to virtualised appliances that are cheaper and more flexible In the field of networking, those approaches also have great potential. This work will focus on leveraging virtualisation and software-defined networking for academic and data center-related purposes. We present a system to simplify the education of students and a system to allow testing and evaluation of new networking protocols in a virtualised environment without the need for real hardware apart from a virtualisation platform. For data center operators, we present a system to leverage network virtualisation without the need for expensive appliances for network functions. All technologies in this work are not only scoped for networking specialists but are also usable by engineers with only general knowledge of the networking field. The procedures and technologies explained in this work lead to scientific work that has been published at different conferences [1–5]

Contents

1 Introduction **1**

1.1 Thesis Outline 1

1.2 Scientific Contribution 2

2 Background **3**

2.1 Virtualisation 3

2.1.1 Advantages of Virtualisation 4

2.1.2 Hypervisors 5

2.1.3 Software-Based Virtualisation 7

2.1.4 Virtualisation Tools 14

2.2 Overlay Routing 15

2.2.1 Principles 18

2.2.2 MPLS 19

2.3 Hybrid Access 22

2.3.1 Hybrid Access principles 22

2.3.2 Related Work 24

2.4 Software-Defined Networking 27

2.4.1 SDN Principles 27

2.4.2 Northbound Interface 29

2.4.3 Southbound Interface 30

2.4.4 OpenFlow 31

2.4.5 P4 32

2.4.6 Network Function Virtualisation / Service Function Chaining 33

3 Testbeds **37**
3.1 VITO . . . 37
3.1.1 Motivation . . . 37
3.1.2 Testbed orchestrator . . . 39
3.1.3 Link modelling . . . 42
3.1.4 CI/CD . . . 48
3.2 IP . . . 48
3.2.1 Concept . . . 51
3.2.2 A Lab Orchestrator for Semi-Virtualized Testbed clusters (LOST) . . . 62
3.2.3 Implementation . . . 65

4 Hybrid Access **71**
4.1 Load-balancing Concepts for Heterogenous Links . . . 71
4.1.1 Performance Comparison of MPTCP and PBLB . . . 72
4.1.2 Flow-Based Load Balancing . . . 76
4.2 Performance Evaluation of FBLB . . . 81

5 Software Defined Networking **93**
5.1 Architecture of P4-SFC . . . 93
5.1.1 Implementation of the SFC Ingress Node . . . 93
5.1.2 Transparent and Efficient VNF Integration on Hosts . . . 96
5.1.3 P4-SFC Orchestrator . . . 97

6 Conclusion **101**

Acronyms **103**

Bibliography and References **107**

1 Introduction

In this section I will outline the contents of my thesis as well as the scientific contribution that can be drawn from it.

1.1 Thesis Outline

The Internet, and any large network, is in a state of constant change. New technologies are developed on a daily basis to cope with the ever increasing amount of traffic and users as well as threats and resilience issues. However, new approaches and technologies are always a part of a bigger system and need to work as such. While theoretical design and validation in a simulation is a feasible way for new technologies to test them on their own, it can't replace tests with real protocol and network stack implementations. Those are written and optimized, sometimes over the course of several years, and may divert from the behaviour one would expect them to show. The biggest example of this are the TCP implementations in the Windows and Linux kernel, which are loosely based on the standards they implement but are not well documented. This work will present a framework that uses virtualization technologies to allow for virtual networking testbeds that still involve the same software used for the majority of servers on the Internet, while also providing accurate link modelling for experiments. The framework, VITO, can be used to fully automate the testing process for rapid prototyping, which will be shown with the example of Hybrid Access. The flexible foundation of VITO will also be shown in a modified version to aid in hands-on networking

classes and in a version that enables data-center operators to utilize NFV[1] and SFC[2] technologies with their legacy infrastructure.

1.2 Scientific Contribution

The scientific contribution of this thesis consists of multiple parts. It is made up of contributions in the field of virtualization, overlay routing, and novel SDN technologies. The main contribution however are not these approaches themself but the combination of those technologies to I created a framework for rapid prototyping of new technologies and will present a case study of a future Internet technology, flow based hybrid access, that was developed and tested using this tool. A cost effective approach for teaching students the principles of networks is built on the core of the prototyping framework which allows for flexible network connections that can be leveraged to conduct experiments on variable network topologies. All contributions are combined in an approach that allows to replace expensive traditional network appliances like firewalls with cost effective off-the-shelf hardware in an overlay routing and virtualization-based multi-tenant NFV/SFC solution that does not require knowledge of networking or virtualization technologies and can be used in conjunction with traditional infrastructure.

[1] network function virtualization
[2] Service Function Chaining

2 Background

This chapter introduces the technical background of this monograph. Main parts of the thesis build upon the field virtualization, hybrid access and SDN, in particular P4.

2.1 Virtualisation

The main goal of virtualisation is to fully leverage the computing power of computers – and in particular servers – efficiently. This can be achieved by sharing the resources of a computer by so-called VMs[1]. Contrary to popular belief, VMs are not used to provide greater flexibility in productive environments. The handling of bare-metal servers and VMs are mostly the same and share the same problems when it comes to provisioning, the need for overprovisioning and maintenance. To provide flexibility, containers and container orchestration platforms are used as they have a significantly smaller footprint than VMs. They will be explained later. VMs are most often used as nodes for such container orchestration platforms. A VM runs on a host just like any normal program, managed by a host OS. In contrast to a normal program, the host OS[2] needs to provide not only resources for the VM but also an entire system platform that resembles the hardware otherwise present in a physical system. Even automatic creation of VMs in cloud environments only differs in speed from the traditional addition of bare-metal hardware to a system. The VM therefore mostly behaves like an actual

[1] virtual machines
[2] operating system

physical system. The host has to ensure that multiple VMs can run isolated from each other in the same environment.

2.1.1 Advantages of Virtualisation

With VMs, sharing resources on a single host is possible with little to no performance loss when using hardware accelerated virtualisation. Without virtualisation, different services that have to be separated –for security reasons, for example – have to run on separate dedicated machines. This, in many cases, leads to a situation where not all resources are fully leveraged as not all services use the full resources of their dedicated hardware such as the CPU[3], RAM[4] or storage. Virtualisation allows different VMs to run on the same hardware without influencing each other while still sharing the hardware resources among them. With this principle, virtualisation reduces CAPEX[5] and OPEX[6], as the amount of hardware needed is reduced, since the hardware resources of a host machine that formerly would have been unused can be used for additional VMs. The isolation also benefits security in two ways. The logical isolation between VMs allows for service separation with hardware support. It also provides security: if malicious software is contained in one VM, it cannot spread out to other VMs. In short, a large part of security can be outsourced to well-tested hardware and does not need to rely on software solutions.

Typically multiple VMs run on a single server. As each VM has different requirements regarding resources, it is mandatory that VMs be distributed to servers according to their needs, to allow optimal resource utilisation. This allows a mix of different services to run on a single server and still meet all requirements with a reduced waste of resources. For example, a firewall has a high CPU requirement while using hardly any storage. A backup server needs storage but very few CPU resources. On a virtualised host it is possible to combine both VMs on a single

[3]central processing unit
[4]random access memory
[5]https://en.oxforddictionaries.com/definition/capes
[6]https://en.oxforddictionaries.com/definition/opex

server to fully leverage all resources. This allows for a more efficient use of the available physical resources.

VMs are not bound to certain hardware. Migration of VMs between multiple physical servers is possible, in most cases transparent for the application running inside the VM. This allows for easier maintenance, since services can be moved to another server during the maintenance and migrated back afterwards. This process holds its own risks and challenges, such as for time-sensitive applications relying on the OS clock, which can get out of sync during migration, but it still provides an easier solution than an active/passive setup or one that needs downtime for hardware maintenance. The same holds true in cases of hardware failure, which opens up the possibility of a nearly instant failure of services. This process may not always work seamlessly, but it can reduce downtime to several seconds instead of several minutes or even hours, in the case of dedicated hardware. All modern VM orchestrators have a feature which enables migration of VMs with a downtime of normally less than 5 s, vMotion by VMWare [8] being one example. Also it is easier to increase resources dedicated to a VM than it would be on a physical host. If a VM needs more resources than the server can provide, one option is to migrate this VM to another server with more resources, allowing for a seamless upgrade without significant downtime. In general, overcommitting the resources of a server is also allowed. While the maximum amount of memory available is a hard limit, it can still be distributed load dependent to allow multiple VMs to have more resources as long as the overall consumption does not exceed the physical hardware constraints. Distributing more CPU cores than physically available is not a technical problem, but if done without care it leads to worse performance for all VMs.

2.1.2 Hypervisors

Typically, a VM runs just like any other user application on a host OS. The difference is that a normal user application does not need an entire system platform. But this part is what makes a VM behave like a physical machine that allows for

the execution of an entire operating system. The VM should not have full access to all host resources for security reasons. For example, a malicious VM might consume all host resources, preventing other services from working. To ensure these principles, some kind of manager needs to be set in place that provides and limits the resources assigned to a VM. In some cases, access to real IO[7] devices, such as PCI cards, has to be granted to a VM which also has to be regulated. The manager responsible for those tasks is called hypervisor.

Hypervisors are classified into two types, Type 1 and Type 2. A type 1 hypervisor runs on bare-metal hardware without an OS to execute it. These are called native or bare-metal hypervisors and are the earliest type of hypervisor, developed as early as 1960 by IBM. The best-known type 1 hypervisors are IBM z/VM and VMware ESXi. They either provide their own configuration interface to manage VMs or require a privileged management VM which is only used to manage the hypervisor, with no other services running on this VM. Type 2 hypervisors are hosted as normal applications under normal OSes like Linux or Windows. These programs provide an abstraction layer for the guest operating system from the host OS. The best-known type 2 hypervisors are QEMU, VMware Player and VirtualBox. Most type 2 hypervisors include a configuration interface, either graphical or text based, and most also include an interface for external control with frameworks like libvirt [9]. The widespread KVM[8], used by the Linux kernel, is some kind of intermediate form. It is part of the Linux kernel and therefore runs directly on hardware, but it requires the normal Linux kernel for resource access and regulation and an emulator like QEMU to run a guest VM. Figure 2.1 provides a comparison between both types of hypervisors. Hypervisors need to manage and enforce access and resource limitations. They are also called VMMs[9] since they are responsible for managing and monitoring VMs at all times. When a VM demands access to protected resources, a trap to the VMM is required so that the VMM can check whether the VM has the permission to perform the requested

[7] Input/Output

[8] kernel virtual machine monitor

[9] Virtual Machine Monitors

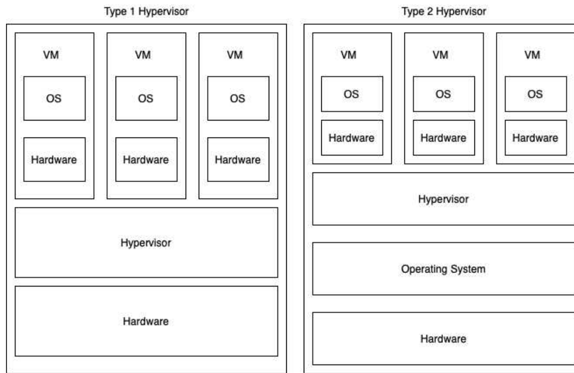


Figure 2.1: *Comparison between Type 1 and Type 2 hypervisors.*

action, and the VMM performs the corresponding syscall to actually execute it. To reduce the overhead introduced by these traps, newer virtualisation extensions try to implement the required features in hardware to increase performance.

2.1.3 Software-Based Virtualisation

Software-based virtualisation can be divided into full virtualisation, paravirtualisation and container-based virtualisation. The first two types share a similar concept, while container-based virtualisation is a special kind of virtualisation that will be explained later.

2.1.3.1 Full Virtualisation

In a fully virtualised environment, the guests are unaware that they are virtualised since a fully emulated system is presented to the VM by the hypervisor. This allows for every guest OS to be run without the need for changes like an adapted kernel. As all resources are emulated, the VMs can run entirely unprivileged, the same as any other application. The resources are exclusively assigned by the hy-

pervisor. The emulation includes three different aspects. All peripheral devices, like input devices or monitors, and storage is emulated by the hypervisor. Memory access and access to shadow pages using an MMU[10] in hardware have to be implemented by the hypervisor in software. All CPU instructions within the VM have to be translated to the corresponding host CPU instruction by the hypervisor. If the CPU architectures on the host and the guest operating system are the same, the instructions need to be translated from unprivileged ring 3/1 instructions to privileged ring 0 instructions. If the CPU architecture differs between host and guest, the hypervisor needs to perform a binary translation between the different machine commands. This usually leads to poor performance, as most instructions cannot be translated in a 1:1 fashion and the translation to corresponding host CPU instructions has to be done by the software. However full virtualisation is still used for cross virtualisation, for example, to emulate an ARM-based VM to emulate a smartphone on an x86 CPU. This is mostly used to develop smartphone software to be run on Android or iOS.

2.1.3.2 Paravirtualisation

In a paravirtualised environment, the VM ‘knows’ that it is virtualised, compared to a fully virtualised VM, where the guest is not aware it is being virtualised. Typically, only the guest OS kernel needs to be aware of being virtualised; the applications running inside the VMs do not. Even within the kernel only the drivers of components that are paravirtualised need to be aware, since typically not all components in a system are paravirtualised. This is because the drivers for the emulated and paravirtualised components need to collaborate with the hypervisor to increase performance, compared to a full virtualisation, in which the drivers need to be emulated as well. While the hypervisor in a fully virtualised environment only receives the commands that would have been sent to the real hardware, a paravirtualised device communicates on a higher layer, allowing the hypervisor to translate the request itself. To allow this functionality, the drivers

[10]memory management unit

for the paravirtualised devices need to be modified to support this kind of virtualisation.

The best-known example of paravirtualisation is XEN [10]. Paravirtualisation is also used in the Linux context with virtio [11, 12]. It provides an abstraction layer over a set of devices as paravirtualised devices and provides a common frontend for common devices like NICs[11], storage or video cards. It is much easier to develop drivers for these paravirtualised devices due to the standardised interfaces. For example, if the VM is able to use a virtio NIC, the OS uses an optimised driver. This driver performs operations to transfer data from the VM to the host in memory to accelerate the network throughput [13].

2.1.3.3 Container-based virtualisation

Container-based virtualisation is a misleading term, as it does not count as real virtualisation. In a container environment only the 'userland', all the software running on a computer except the kernel, of the guest OS is executed. The Linux kernel allows separation of processes using a system called cgroups [14]. This system allows for different processes on the same host to run separated from each other with a defined resource scope. The same principle holds true for network namespaces [15], allowing for different networking stacks on the same host. A container is a process, running in its own networking namespace, isolated with cgroups, running either a single process (e.g. with Docker [16]) or a whole userland of a Linux distribution (e.g. using lxc [17]). This is done by using the whole filesystem and init system of the guest system and executing them as a normal process. For the user, this construct looks like a full operating system. In both cases, all processes share the same host kernel, leading to a smaller overhead than full or paravirtualisation, since only a single kernel needs to be booted for all services, compared to one kernel per service. Container-based virtualisation does not need any hardware support at all; all functions needed are present in the Linux kernel.

[11] network interface cards

2.1.3.4 Hardware-Assisted Virtualisation of the x86 Architecture

Unlike other CPU architectures, x86 is not natively virtualisable in hardware. Therefore, other hardware architectures, like POWER or ARM, are traditionally used for virtualisation. However, Intel and AMD began to add extensions for virtualisation, including system calls and special registers, to their architectures around 2007. The main goal is to reduce the huge performance loss introduced by software virtualisation and to provide near native performance for VMs. Extensions to the basic architecture, enabling further virtualisation of, for example, PCI[12] devices, were introduced. In this chapter the Intel naming scheme is used as those processors are still more common in server environments, and all experiments and technologies presented in this work were developed with Intel CPUs. However, all extensions presented have an AMD equivalent providing the same functionality.

2.1.3.5 Extensions to the x86 Base Architecture

The first extension introduced to enable hardware-based x86 hardware virtualisation is called VT[13]-x [18]. It introduces an additional privilege system and special CPU registers. It allows the guest to enter and exit a special virtual execution mode. The virtual execution mode presents itself to the guest as running with full privileges while the host OS is still protected. To allow memory virtualisation, hardware-based shadowing in the MMU is introduced. While page tables map the address of virtual memory used by actual programs to physical addresses, shadow tables are pseudo-page tables that allow mapping between the virtual memory of the VM and the physical address on the host. Without VT-x this functionality needs to be implemented by software, with poor performance.

Interrupts within a VM typically cannot be processed directly by the VM. Interrupt processing requires that the guest leave the context of the VM and switch to the context of the host. After the interrupt is handled, the context is switched

[12] Peripheral Component Interconnect

[13] Intel virtualization technology

back to that of the VM. Obviously, context switches are complicated and time-consuming. To reduce this overhead, APICv[14] [19] emulates access to an advanced programmable interrupt controller and interrupt processing. By doing this, APICv eliminates around half of the context switches, which significantly improves performance.

The virtual memory of the host system serves as physical emulated memory for the VM. The translation of physical memory to virtual memory is done by the MMU. However, in data centres, VMs may serve as hypervisors for other VMs themselves. This is called nested virtualisation and allows a VM to run inside a VM. The inner VM also needs to translate its physical addresses, which are already virtualised by the MMU to virtual addresses for the inner VM. This process must be done by the software and is also time-consuming. SLAT[15], also known as nested paging or EPT[16] [20], is an extension to the MMU providing hardware-assisted inner address translation. With EPT, every guest's physical address is treated like a host virtual address, as with normal applications. According to a VMware evaluation paper [21], this increases performance in MMU intensive benchmarks up to 48%. For MMU-intensive micro benchmarks, it performs as much as seven times faster than a traditional system. EPT is also needed to launch a logical processor directly in real mode. Intel calls this 'unrestricted guest' mode.

2.1.3.6 I/O MMU Virtualisation

As already introduced, memory can be distinguished between the actual physical memory and the virtual memory presented to the processes. Each process perceives that it can use the entire memory of a system and can freely choose the addresses. The MMU is responsible for translating between virtual addresses and physical addresses.

[14]Advanced Programmable Interrupt Controller virtualization
[15]Second Level Address Translation
[16]Extended Page Tables

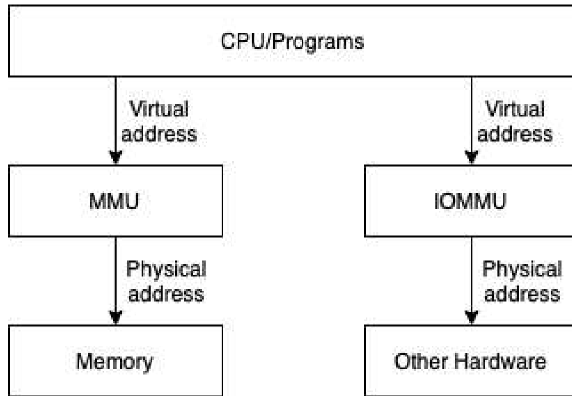


Figure 2.2: *The IOMMU.*

The same principle is applicable for hardware devices. The devices or I/O[17] addresses are presented as virtual addresses and therefore have to be mapped to hardware addresses. The component achieving this task is called the IOMMU[18]. Figure 2.2 shows the analogy between an MMU and an IOMMU. The IOMMU is called VT-d by Intel [22]. The IOMMU is responsible for device mapping, for DMA[19] access to the device and for remapping interrupts. Some IOMMUs may also include memory protection mechanisms. An IOMMU always needs additional support in the chipset and the firmware, such as the BIOS[20], as it cannot work on its own.

Most often VT-d is used to pass through physical devices from the host system to a VM, such as NICs. This grants exclusive access to a physical device for a VM. The host no longer sees the device. This mechanism does not need any software or virtualisation layer support, so there is basically no performance

[17] input/output
[18] I/O memory management unit
[19] direct memory access
[20] Basic Input/Output System

difference between using a device on the host or inside a VM. The actual address mapping is achieved via peripheral memory paging with the help of the ATS[21] and the PRI[22].

The IOMMU may also separate the device into different groups to enhance security. Only devices within the same group can communicate with and influence each other. Neither read nor write access is allowed outside the own group. However, this is not present on all systems as it requires the ACS[23] [23] feature.

2.1.3.7 Network Virtualisation

Most VMs are placed inside data centres or provide services that require dedicated network access. As network speed nowadays is quite high, with 10 Gb/s per machine in a data centre being common, a key goal is good performance and efficiency of the networking virtualisation. In contrast to SDN[24], where network virtualisation is related to network slicing, in this context network virtualisation describes the virtualisation of network devices and pass-through to VMs. Intel summarises the technologies used for networking virtualisation under the term VT-c [24]. Compared to the other technologies presented, VT-c not only needs to be present in the CPU but also needs support from the networking device, the firmware and the chipset itself.

VMD-q[25] [25] is a technology that enables multiple physical queues in a single NIC in hardware. Those queues can be used for different purposes and are connected to an internal switch within the NIC. For example, they can be used to implement QoS[26] by separating different traffic classes in different queues and processing them differently. The queues can also be used for network virtualisation. In this case, each queue is associated with a virtualised NIC and implements

[21] Address Translation Service
[22] Page Request Interface
[23] Access Control Services
[24] software-defined networking
[25] virtual machine device queues
[26] Quality of Service

packet forwarding to the VMs in the hardware.

SR-IOV[27] [26] is an extension to the PCIe[28] standard. It provides a standard to distinguish between two kinds of PCI devices. PFs[29] are normal PCI devices that are directly connected to the PCI bus and are able to operate without additional hardware. Some PFs can host virtual lightweight PCI devices that cannot run on their own without the PF. Those devices are called VFs[30]. The PF manages and shares some registers with the VF. For example, a NIC is a PF with one or more physical network ports. If it hosts VFs, they do not have their own networking ports and need access to the PF queues. Figure 2.3 compares different network virtualisation approaches. The first one shows a software based emulation, the second one leverages VMD-q and the third one uses VMD-q in combination with SR-IOV. This combination allows the instantiation of multiple VFs, each associated with a dedicated queue. Each VF can be individually passed through to a VM. Such a virtualised NIC provides around 95% of native network bandwidth.

2.1.4 Virtualisation Tools

All testbeds and experiments presented in this work run on Linux-based OSes. In this section, the three most common virtualisation tools and the used container technologies are described.

The hypervisor already present in the Linux kernel is KVM [27]. KVM is therefore always available on the host system; no additional overhead introduced by a hypervisor, such as VirtualBox, is required. As part of the Linux kernel, KVM has direct access to the hardware available on the system and can control access by the VMs. KVM supports all available virtualisation extensions of the x86 platform and achieves good performance. However, KVM is not implemented to run on its own; it is usually used with an emulator like QEMU [28].

[27] single root i/o virtualization
[28] Peripheral Component Interconnect Express
[29] physical functions
[30] virtual functions

QEMU is a general-purpose emulator that is widely used on the Linux platform. Even though it is able to emulate all kinds of hardware and architectures, we only utilise the x86 virtualisation, the paravirtualised virtio devices described earlier and emulated network adapters. QEMU is able to use KVM as its virtualisation backend and therefore benefit from the hardware-assisted virtualisation features of KVM. Throughout this thesis, if not described otherwise, all storage and NIC devices are paravirtualised using virtio, and the whole VM is virtualised with the assistance of KVM.

QEMU itself is a powerful tool, but its direct usage is discouraged. The parametrisation of QEMU is complex as it is based on a list of command-line parameters to define the hardware and properties of a VM, which makes the usage tedious and error prone. For that reason, there are different tools relying on QEMU as a virtualisation tool but enabling a simplified management of VMs. In this work we always used the framework libvirt [9], which is the most widespread; it provides VM definitions in XML[31] format which can be defined or edited either via CLI[32] or with a GUI[33] called virt-manager. The CLI is called virsh and can be used to modify VM definitions and even run VMs. The XML definition is well formed and human readable, allowing easy editing. In addition, it also allows easy cloning and porting of VMs, since only the definition and the hard disk file are needed.

2.2 Overlay Routing

In this section the principles of overlay routing will be described. The term overlay routing is very widespread; we will therefore focus only on the parts that are relevant for this work.

[31] eXtensible Markup Language
[32] command line interface
[33] graphical user interface

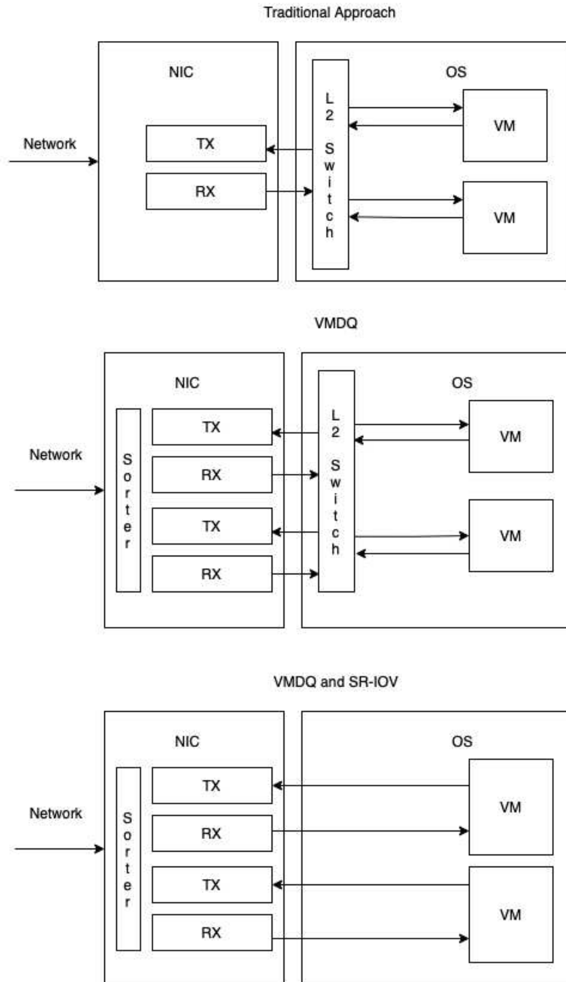


Figure 2.3: *Different approaches to network virtualisation, including VMDQ and SR-IOV.*

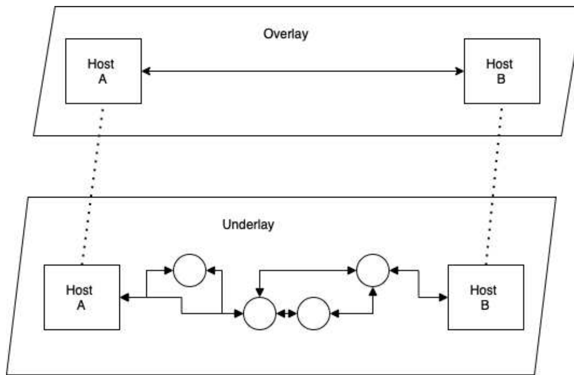


Figure 2.4: *High-level abstraction of the overlay network principle.*

2.2.1 Principles

The term overlay describes a logical network that spans a physical network but does not share the underlying topology. Therefore, overlay routing can be seen as a kind of abstraction layer for physical networks. Figure 2.4 describes the basic principle.

The lowest layer is the physical network with real existing connections and hardware devices. On top there is a new layer representing the logical overlay network which utilises the physical network for packet transmission but can also present paths that are not part of that physical network to the operator. Those paths will, if chosen, be translated into a sequence of hops in the underlying physical network. While the actual packet forwarding still has to be done on the physical topology, only the logical topology is presented to the user and administrator. The translation between a logical path and a physical path in the network has to be done with the help of an overlay routing protocol like MPLS[34] [29] or LISP[35] [30].

Using a logical network instead of the physical one allows simplification of several use cases. A virtual full mesh allows every other device to be reached with a single hop, effectively eliminating all routing complexity from the devices, shifting them to the underlying physical network. Overlay networks can also be used to separate networks from each other. A physical network can be split into several logical networks, allowing a multitenant system with the same underlying infrastructure or a separation of an infrastructure into multiple security zones. In this case, the network operator can define different logical topologies for the customer without the need for changes to the current infrastructure. A third application for overlay routing is traffic engineering and resilience. A logical path in an overlay network can be mapped to any physical path in the underlying network. This feature can be used for traffic engineering; the network operator can specify the path the packet takes through the net, even if the destination and the

[34] MultiProtocol Label Switching

[35] Locator/Identified Separation Protocol

logical path always stay the same. This principle can also be utilised to protect the network from link or node failures. If a link or a node fails, there is no change in the overlay network, and the logical path stays the same. On the physical network, however, the packets can be switched or routed over a different path to the destination. This feature can hide failures in the physical network from the logical network by simply rerouting packets in a transparent fashion for the application.

However, this flexibility comes at the price of additional signalling overhead, which can also lead to lower performance, as each packet requires more processing at each hop. To leverage overlay routing, an overlay routing protocol is required, adding at least one additional header. Providing realistic link costs to the overlay network requires additional signalling and hardware support. For real-time or latency-sensitive applications, this leads to an additional overhead and may render this solution too costly compared to a less flexible physical network connection.

In the following we will describe MPLS as an example of overlay routing and switching.

As the field is too widespread for this thesis, MPLS is chosen as an example of the principles because it was used in the contributions to this thesis. Other overlay routing protocols may work on different layers of the ISO/OSI model and therefore may be suited for different use-cases with different benefits and limitations. Comparing them is not part of this work.

2.2.2 MPLS

MPLS is a label-switching protocol located between ISO/OSI Layers 2 and 3. It works by assigning labels to paths within the networks and then using these labels for forwarding decisions. Figure 2.5 shows a typical MPLS network. Each packet gets an MPLS header with the label of the desired path, such as the destination or the egress of the MPLS network. An MPLS switch has a table with all available combinations of ingress port and label and the corresponding actions. These actions are assigning a new label and forwarding it to an egress port. An

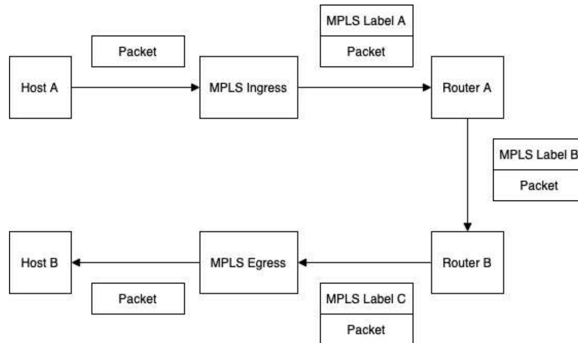


Figure 2.5: *Packet traversal in an MPLS enabled network.*

MPLS switch that receives this packet compares the label and the ingress port with its own label table and then switches the label with a new one in the matching table entry and sends the packet out to the port specified in the entry. This reduces the switching complexity inside the hardware devices since the amount of possible forwarding decisions is limited, but it still allows the switching decision to be based not only on the destination, as in traditional networks, but also on the source, since the ingress port and label are taken into consideration. As shown in Figure 2.5, a packet to the same destination can be routed over different paths depending on the ingress port, even if both packets have the same destination label. This can be used, for example, for implementation of QoS classes in which some packets are allowed to take a better path than others. One way to use MPLS – as done by, for example, Deutsche Telekom [31] – is to assign one label to each node. Effectively, this leads to a full mesh of tunnels that allows the packets to reach every other node with a single label.

MPLS does not need to be used as an overlay network protocol; operation as a normal switching protocol is possible and will be demonstrated later in this work. It is possible to assign one label to each path and present the whole physical network to the user. In such a scenario, there are two options on how to operate the

MPLS network. Source routing is done on the switches by checking the ingress port and ingress label and switching said label with the one for the next segment, based on where the packet came from. For traffic engineering purposes, the path through the network can be already defined at the source. As every segment is specified by a label, it is possible to push a whole stack of those labels in the order the segments need to be traversed to the packet. Each switch then only needs to pop the uppermost label from the stack and knows the segment to which the packet needs to be forwarded. This approach allows for a more efficient switching approach at the cost of larger packet overhead. When using label stacking, only the ingress router needs to push labels on the stack, and combining this with an SDN solution that permanently updates the best paths of the network allows for optimal routing while only the ingress switches need to be updated if paths change.

If used as an overlay network, MPLS can be used to create a virtual full-mesh network which shows a connection from every switch to every other switch with a single hop and therefore only a single label. Those labels represent tunnels in the network which are not always direct connections, but they most often reach over several physical hops. Labels may be exchanged on the way by other switches. However, for the sending host, the path looks like a single segment.

2.2.2.1 Segment Routing

With label switching, an MPLS label identifies a connection. The ingress label-switching router (LSR) pushes a label onto a packet, intermediate LSRs switch the label according to their forwarding tables and the egress LSR pops the label. Segment routing (SR) is a new approach for source routing and may leverage MPLS forwarding. Here, a label identifies a segment, which may be a link, a path, a node or something else. The ingress LSR pushes a label stack onto a packet. LSRs forward the packet according to the topmost label and possibly pop it. Thus, with SR, the network can remain unaware of individual connections, as only ingress LSRs need to know them to push the right label stack. However, most MPLS nodes can push only a few labels. In this work, we utilise SR and

program a P4-capable switch for pushing large label stacks.

2.3 Hybrid Access

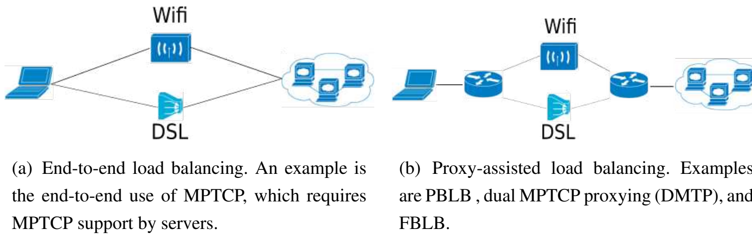
This section describes Hybrid Access. A set of technologies that are used to leverage multiple access lines for internet access to overall improve the service for users. While end-users can profit from a higher access speed to the internet by bundling two or more access methods, business can reach a higher resilience while leveraging existing infrastructure. This chapter is based mostly on a collaboratively written unpublished technical report.

2.3.1 Hybrid Access principles

With multi homing, a network is connected via multiple links to the Internet. This reduces its risk to be disconnected in case of a failure and improves the reliability of its Internet connectivity. With hybrid access (HA), a user's home gateway (HG) is connected over multiple access technologies, e.g., 5G, LTE[36], DSL[37], or cable, to the BNG[38] which is also called broadband remote access server (BRAS). Thus, HA is a special case of multi homing. Its peculiarity is that load balancers at the HG and at the BNG may distribute traffic over all available links to increase the access bandwidth.

Load balancing methods can be classified into end-to-end solutions and proxy-based solutions. With end-to-end solutions, endpoints load-balance traffic over multiple path. MPTCP[39] is an example [32]. End-to-end load balancing requires that various paths are visible between the endpoints and that both endpoints support the load balancing method. It may be applied between, e.g., a multi-homed smartphone and a server supporting MPTCP (see Figure 2.6(a)). It is not applicable for endpoints behind a HG that have only a single IP interface towards the HG

[36] Long-Term Evolution
[37] digital subscriber line
[38] Border Network Gateway
[39] Multipath TCP



(a) End-to-end load balancing. An example is the end-to-end use of MPTCP, which requires MPTCP support by servers.

(b) Proxy-assisted load balancing. Examples are PBLB , dual MPTCP proxying (DMTP), and FBLB.

Figure 2.6: *Load balancing variants.*

because they see only a single path to resources in the Internet. It is neither applicable if the contacted server does not support MPTCP. As the majority of servers does not yet support MPTCP, end-to-end load balancing cannot be effective, yet, so that other solutions are needed.

With proxy-assisted load balancing, a load balancer (LB) load-balances the traffic over available paths at an intermediate hop where these paths are visible, e.g., at the HG and the BNG (see Figure 2.6(b)). Depending on the specific load balancing approach, a recombination function (RF) may restore the original traffic, e.g., remove potential load balancing headers or restore the correct packet order. In a HA context, LBs and RFs can be deployed at HGs and BNGs. Proxy-assisted load balancing is most suitable for HA as it does not require cooperation from endpoints.

There are various options for proxy-based load balancing. Packet-based load balancing (PBLB) distributes traffic over links without taking flow information into account, i.e., packets from the same flow may be balanced over different paths. Thus, the RF must restore the original packet order. Flow-based load balancing (FBLB) distributes traffic such that all packets of a single flow are carried over the same path. Here, the RF may be simple. The third category is applicable to connection-oriented traffic only, e.g., to TCP. The LB terminates single-path TCP connections and reestablishes an MPTCP connection towards a RF. From

there, the traffic is further relayed to the destination within normal, single-path TCP connection. We call this variant dual MPTCP proxying (DMTP).

Load balancing is not a new challenge. However, most existing approaches balance traffic over identical, parallel links, mostly on layer 2, which is also called bonding. Load balancing for HA is more challenging as the characteristics of the bundled links may be heterogeneous. This may lead to performance problems for TCP when PBLP is used for HA. DMTP is applicable only to TCP traffic and it is extensible only to connection-oriented traffic. While PBLB [33, 34] and DMTP [35] have been recently considered, there are no applicable algorithms and comprehensive performance evaluations for FBLB.

2.3.2 Related Work

The survey in [36] provides a general overview on load distribution over multi path networks. In contrast to that work, we consider the impact of load balancing on congestion controlled traffic. In the following, we provide an overview of load balancing mechanisms that have been proposed and possibly deployed on different layers.

2.3.2.1 Load Balancing on the Link Layer (L2)

Load balancing algorithms on the link layer schedule L2 frames to available links and recombine them afterwards. They may even split them and transmit the parts over all links. Mostly homogeneous links are needed. These approaches are limited to a single hop. Therefore, they cannot support HA.

The PPP Multi link Protocol (MP) [37] is an extension to PPP[42]. It combines several physical L2 links to a single logical link. L3 packets may be fragmented and fragments can be sent over different links that are established via PPP authentication and LCP[43]. The fragments are buffered at the receiver and the L3 packet is reassembled. MP has been used, e.g., for ISDN channel bonding.

[42] Point to Point Protocol
[43] Link Control Protocol

The Link Aggregation Control Protocol (LACP [38], 802.1ax) is designed for bandwidth aggregation on L2 and L3. It is designed to bundle point-to-point full duplex links of the same speed, but does not work with heterogeneous links. LACP supports signaling between all endpoints, which enables dynamic trunking options, allowing for a variable amount of interfaces with the same characteristics to be bundled.

Hari et al. proposed “strIPe” [39] for combining logical channels on any layer which implements FIFO functionalities, allowing to ensure packet order during the process.. It continuously synchronizes sender and receiver and requires knowledge of the link characteristics. For optimal load balancing those conditions should be stable. It is not suitable for wireless links whose link characteristics vary over time. The paper uses the term "quasi-FIFO" to describe periods where synchronization between both endpoints is lost due to link characteristic changes or packet loss. During these periods packet reordering may occur.

2.3.2.2 Load Balancing on the Network Layer (L3)

Load balancing algorithms on the network layer distribute IP packets over different paths which may have heterogeneous or even varying characteristics. Therefore, consecutive packets carried over different links may arrive out-of-order so that a recombination buffer is needed to delay the packets carried over the faster path. In Section 4.1.1 we show that PBLB may cause performance issues for TCP flows if the paths have significantly different delay.

Bonding of GRE tunnels [40] is described in RFC 8157 [33]. The document describes requirements, header format as well as signaling for load balancing support. The RFC proposes packet coloring based on [41], which is based on a token bucket, with subsequent color-based load balancing. It implements a cheapest-pipe-first approach: traffic is sent of a preferred link and only traffic exceeding its capacity is spilled over to a non-preferred link.

LISP Hybrid Access [34] provides a protocol framework for dynamic load-balancing traffic over the Internet using the Locator Identifier Separation Protocol (LISP [30]). It does not propose specific algorithms for LB and RF as they are

not specific to the overlay network and should be considered independent of its usage.

Equal-cost multi path (ECMP) is a load balancing method on L3 which takes flow information into account so that packets of a single flow are carried over the same path. In [42], the performance of ECMP has been investigated and in [43] the load balancing result has been studied for several stages of load balancing.

FBLB load-balances flows over different paths. In contrast to ECMP, the load balancing result depends on flow rates and available bandwidths. On the one hand, this obsoletes the need for a complex RF as packets are not reordered. On the other hand, a single flow cannot benefit from the capacity of several access links. Thus, the approach is simple, likely to perform well only in the presence of sufficiently many flows. We are not aware of any definitions or performance evaluations of FBLB.

2.3.2.3 Load Balancing on the Transport Layer (L4)

Load balancing algorithms on the transport layer distribute L4 segments over different paths.

Multi path TCP (MPTCP [32]) describes a TCP extension for multi path operation. It associates multiple IP addresses with client and server and the combination of these tuples defines multiple subflows. A scheduler distributes a TCP connection's traffic over these subflows and each subflow has its own congestion control. Thereby, MPTCP load-balances traffic over all visible links. The major purpose of MPTCP is mobility support on layer 4. While a normal TCP connection is bound to a single client and a single server IP address, MPTCP can add new addresses so that a connection survives when a user roams into another network. Apple's iOS supports MPTCP and Siri is a well-known application leveraging MPTCP. MPTCP is part of experimental Linux kernels but not part of the mainline Linux kernel (15.11.2019). Therefore, it is not widely deployed. Moreover, MPTCP requires support from both TCP client and TCP server. An MPTCP proxy has been proposed in an expired Internet draft [35] and may be used by MPTCP clients to communicate with a non-MPTCP-capable server.

MPTCP may be used for end-to-end load balancing. Its effectiveness depends on the availability of MPTCP-capable servers.

MPTCP may also be leveraged for proxy-assisted load balancing in a HA context. That means, end-to-end traffic may be either tunneled or proxied between HG and BNG. Tunneling a TCP flow over MPTCP raises performance concerns as the interaction of the inner and outer congestion control loop is counterproductive. The proxying approach is limited to TCP traffic only; an extension to other connection-oriented traffic seems feasible, an extension to connection less traffic seems difficult.

The Stream Control Transmission Protocol (SCTP [44]) is an extensible transport protocol and supports load balancing [45]. However, SCTP is only marginally deployed. Therefore, it usage for end-to-end load balancing suffers from the same problem as MPTCP. And the same concerns apply for SCTP-based proxy-assisted load balancing as for MPTCP-based proxy-assisted load balancing.

QUIC is another transport protocol on application layer. It is based on UDP, but utilizes the same congestion control as TCP. There are plans to extend QUIC to multipath QUIC [46], but a working implementation is still missing.

2.4 Software-Defined Networking

In the following, we introduce the idea of SDN and provide an overview of the two most popular implementations of modern SDN, OF[44] and P4[45]

2.4.1 SDN Principles

SDN is an approach that increases flexibility and configurability in networking environments and breaks with pre-existing networking paradigms. As the name

[44] OpenFlow
[45] Programming Protocol-Independent Packet Processor

suggests, SDN diverts from the classical approach of hardware-defined networking, where all packet processing is done by physical devices on their own. With SDN, networking devices are viewed as programmable devices that can be influenced from the outside to obtain the desired behaviour. Technologies to program switches and other network devices will be described in this chapter.

While this approach is not new, today's hardware finally has the power to fully leverage these principles and gain advantage despite the additional overhead introduced in the process.

Traditionally, the data plane and the control plane of a networking device, such as a switch, are coupled in a single device, where the control plane is responsible for forwarding decisions and the data plane forwards the packets. The control plane is part of the firmware and therefore most often closed-source. Dedicated protocols, like routing protocols, are used for communication between devices, each device calculates its own forwarding decisions with the help of shortest-path algorithms.

SDN breaks with this principle by separating the control plane from the data plane of the networking device. The control plane is most often a logically centralised SDN controller. An SDN controller has a global view on the network and is able to calculate forwarding tables for all connected devices. The result of these calculations is then pushed to the devices to configure the data plane. A global view allows for better algorithms for path calculation, leading to a more optimal result; however, it requires centralised high computing power. It can also be used to facilitate more advanced mechanisms like traffic engineering. However a global controller also is a single point of failure; therefore, mechanisms for redundancy and load-balancing are a important part of the design. The most used implementation of this concept is the configuration of switches using SNMP[46] [47].

A controller works with network applications that implement the different

[46]Simple Network Management Protocol

functions routers and switches provide, like forwarding based on MAC[47] or IP[48]. Since all calculations are outsourced, a pure SDN switch does not need any intelligence apart from simple forwarding. This allows for the rapid development and deployment of new features, such as traffic engineering mechanisms as controller apps without the need for upgraded SDN hardware.

We do not provide a full explanation of all parts of SDN architecture, as this is not the focus of this work. A more detailed view can be found in surveys like [48] or [49]

2.4.2 Northbound Interface

SDN apps communicate via the NBI[49] with the actual SDN controller. The controller has to provide an API[50] like a direct programming interface, a REST[51] service or even a dedicated communication protocol. The NBI provides an abstraction layer over the controller's view of the network and the possibility to allow configuration changes on a high level. Some controllers do not expose the devices directly and only allow specification of the desired behaviour of the network. Most often a traffic engineering mechanism does not need to know details of the devices used to specify a path through the network; it does not even need to know the exact physical topology. The specifications may be formulated in a higher programming language compiled for the controller using a given NBI, which then translates it to instructions in the SDN protocol used in the network.

No standardised NBIs have been developed as of now. Every controller supports a different set of NBIs using different methods of communication, models and programming languages. Popular examples for programming languages are Frenetic [50], Pyretic [51] and Procera [52], which all provide a declarative syntax based on a functional programming style with recursions. This allows to

[47] media access control
[48] Internet Protocol
[49] northbound interface
[50] application programming interface
[51] representational state transfer

formulate and combine network policies. Another alternative is NetIDE [53]

For prototyping, most often a direct programming interface or an REST API is used to speed up the process and allow easier implementation of new features. Working directly with the controller through an abstraction layer instead also leads to the results being easier to debug. Therefore, most prototypes have no clear separation between the networking app and the controller; both can be run in the same program for easier deployment.

2.4.3 Southbound Interface

The interface between the controller and the actual network hardware is called the SBI[52]. The most popular SBI for that task is OF [54]. There are many other variants like SNMP and NETCONF[53] [55]. Programming languages like P4 [56] could also be considered part of this group as they serve the same purpose as the protocols mentioned above. In 2014, Cisco proposed OpFlex [57] as an alternative to OF and tried to standardise it at the IETF [58]. The other already standardised protocol is NETCONF, which differs from the SDN protocls in this work in that it is more in line with traditional network programming protocols like SNMP. The XML-based YANG [59] is used to model the network and the configuration of networking devices. Each networking device can specify its own YANG[54] model with device-specific variables. The network devices and the controller communicate via XML-based RPCs[55]. Other standardised implementations that are not used in this work include ForCES [60, 61] and SoftRouter [62]. However these protocols are not widespread and are therefore irrelevant to this work. The prototypes built in this work are built using OF and P4.

[52] southbound interface
[53] NETwork CONFiguration protocol
[54] yet another next generation
[55] remote procedure calls

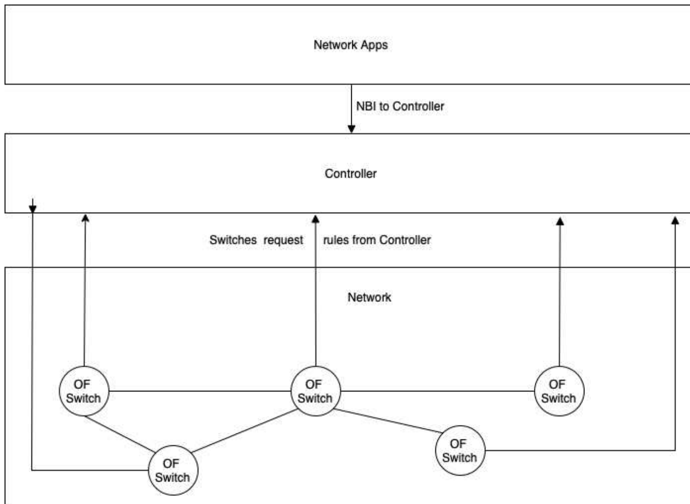


Figure 2.7: *A network with OF enabled switches, a generic controller and non-specified networking aApps as source for the controller rules.*

2.4.4 OpenFlow

OF[56] [54] is one of the two currently most prominent SDN architectures and SBIs. The project originated at Stanford University in 2007 and is maintained by the ONF[57]. The typical architecture consisting of three layers and the interfaces between them is depicted in Figure 2.7. The lowest layer is the infrastructure layer, which includes OF–capable switches that communicate with the controller via the OF protocol in the control layer. The OF controller only provides basic network services. The actual network apps run on top of the controller in the application layer and communicate via an NBI, such as REST with the controller.

The basic idea of OF is to classify packets by certain match rules into different

[56] OpenFlow
[57] Open Networking Foundation

flows. Instruction for each flow are stored in a so-called flow table. Those instructions may include actions like forwarding packets to a certain out port, dropping a packet or forwarding a packet to the controller. The last action is important since a switch may not have applicable flow rules for all kinds of traffic, therefore unknown flows need to be forwarded to the controller which then calculates a rule for these flows. The controller then pushes the new rule back to the switch, which can process subsequent packets on its own.

There are different versions of OF, all supporting a different feature set; however, for this work only a basic feature set of OF is important, and therefore they are not explained in detail. The same holds true for the different controllers available.

2.4.5 P4

P4 is a language that defines the forwarding behaviour of P4-capable networking devices, such as switches and NICs. P4 differs from other approaches in that it lets an application define the matching tables, packet headers and packet operations itself, providing more flexibility for the operator. The language is similar to a simple C program but has a reduced function set, missing such things as loops and recursions to allow it to run on networking hardware. P4 programs are compiled directly for the target platform on which the program runs.

To receive better support from hardware vendors there are different standards which define the P4 packet processing procedure. The most widespread is PISA[58]. The pipeline of a PISA system is depicted in Figure 2.8. It consists of three parts. The parser is responsible for parsing incoming packet headers. It can be described as a finite state automate which can be used to extract packet headers from an incoming byte stream for further processing. The match-action pipeline executes operations on the packet itself; for example, to implement simple IPv4 forwarding, a table lookup for the next hop of the parsed IPv4 address has to be performed. Header fields can be modified in this step. The size and functionality

[58] Protocol Independent Switch Architecture

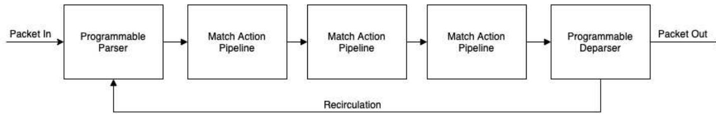


Figure 2.8: *A generic pipeline that follows the PISA standard.*

of the match-action pipeline depends on the hardware implementation. It is also possible that a P4 switch offers so-called *extern objects*, functions that are not directly implemented in P4 but can be accesses via an API, such as checksum calculations which are done by the hardware itself and would be slower when done by software. The last step is the deparser, which serialises the packet for forwarding.

2.4.6 Network Function Virtualisation / Service Function Chaining

NFV is a technology that aims at rebuilding existing hardware components used in traditional networks such as firewalls in software. Those hardware components are often costly and have a fixed performance. Replacing them with software that can run on off-the-shelf hardware makes it possible to increase flexibility and reduce costs. Multiple approaches for this topic have been around for some time, with most involving specialised programming techniques or knowledge of additional networking stacks. A possible approach to realising NFV will be described in this thesis.

2.4.6.1 Protocol Stacks for SFC

The IETF has identified SFC as a problem for traditional networks due to their topological dependencies [63]. Traditional networks have a rather static configuration, but SFC requires a highly dynamic network. Having the potential for constant changes within the network could limit scalability and high availabilty

in the long run.

A major result of the IETF's SFC working group is the network service header (NSH) [64], which consists of three parts: a base header providing information about the header structure and the payload protocol, a service path header containing the path identification and location within a service path and a context header for metadata. A special feature of the NSH is that it can carry metadata among VNFs.

Another document proposes an MPLS-based forwarding plane for SFC [65]. It suggests tuples consisting of an 'SFC context label' and an 'SF label' similar to the NSH. The context label identifies the SFC by the contained service path identifier (SPI), and the SF label identifies the next service function to be actioned. In the case of label switching, the context label is maintained and used by LSRs to switch consecutive SF labels for VNFs. In the case of segment routing, tuples of context/SF labels are stacked by the ingress LSR and consecutively popped with completed VNF operations. A similar approach is described in another working group draft [66].

These protocols are partly competing and not fully compatible. In all proposed protocol suites, all devices involved in an SFC, such as forwarding nodes and NFs, need to be SFC aware; that is, they need to respect protocol specifics. P4-SFC, the approach described in this thesis, allows customers to use VNFs that are not SFC-aware. Furthermore, it leaves the network unaware of SFCs, utilises only common MPLS labels and requires forwarding nodes to pop only single labels; that is, no special hardware features are needed. Only the SFC ingress node pushes a label stack.

2.4.6.2 Selected SFC-Related Activities

The ETSI has published a set of documents describing an architecture for networking operations and orchestration (MANO) of NFVs [67]. It provides an overview with a focus on interoperability, but it does not offer an NFV/SFC networking stack.

The Open Platform for NFV (OPNFV) [68] was started by the Linux Foundation in 2014. It is a cooperative project among 20 companies with the goal of developing an NFV infrastructure (NFVI) software stack to build and test NFV functionality. Its long-term goal is to provide a standard platform for NFVI.

Most commercial cloud operators, such as Amazon [69] or Microsoft [70], offer configurable, complex services to their customers based on NFV/SFC. Examples of such NFs are firewalls, gateways and load balancers. These services are comfortable for customers but are limited to functions provided by cloud operators. P4-SFC allows customers to upload their own VNF binaries.

NFVnice [71] is a user-space scheduler on a host that decides whether a packet is delivered to its desired VNF. It also monitors all VNFs in the system. If VNFs in a later stage of an SFC are overloaded, NFVnice drops packets while they are at an earlier stage of the SFC to reduce wasted work.

P4NFV [72] and P4SC [73] propose to implement NFs based on P4-capable hardware because general-purpose hardware is too slow for fast-packet processing. Their contribution is an architecture for the management of VNFs on P4 switches. This work is applicable to P4-based hardware and software switches.

3 Testbeds

This section contains contributions to the automation of testbeds and a derived work in the form of a novel concept for the Networking Labs at the University of Tübingen.

3.1 VITO

This section describes the VITO[1] which was published as "VITO: Virtual Testbed Orchestration for Automation of Networking Experiments" at VALUETOOLS 2017. It is designed to automate networking experiments in a virtualized environment on a single server. The main intention behind the development is the performance testing and evaluation of novel networking protocols. It relies heavily to Open Source components, most of them present in the Linux kernel. Networking nodes are modelled by virtual machines and the Linux module TC is used to model link characteristics. Both will be explained in detail in the coming sections. An experimental performance analysis gives recommendations for the configuration and provides an application example.

3.1.1 Motivation

In computer networking research, simulation, emulation, and hardware testbeds are used to implement new protocols and control mechanisms, and evaluate their performance. With simulation, control over network properties and topologies is easy and it is convenient to perform large experiments. However, a challenge is

[1] VIrtual Testbed Orchestrator

to correctly model complex protocols like various TCP variants with sufficient accuracy as well as correct sampling and evaluation of the results. Some network simulators allow the integration of real Linux network stacks, but the main flaw is that they are bound to certain operating system versions that are currently not up to date, which is a problem for testing latest protocol enhancements [74]. Furthermore novel technologies should also be tested with the most recent Kernels used in production and not the ones that are provided in an outdated simulator. Highly complex communication technologies may take long simulation times if modelled on a low level. Real time experiments are not possible anymore once multiple Kernels need to be executed as well as a simulation of a physical network. A slightly different but mostly similar approach are network emulators which interconnect real devices over a simulated network which can be easily configured. An example is NetSim [75]. Network emulators generally can support only low networking speeds so that only limited experiments can be conducted. Highspeed experiments are not possible since the simulation of the network is a task which can't be optimized for off the shelf processors. Most true to reality i experimentation on hardware testbeds which allows the application of original protocol stacks and real protocol implementations. However, hardware testbeds are heavy-weight solutions. If an experiment is large in terms of nodes, it requires lots of physical hardware, administration overhead, experimentation is expensive, requires lots of space, energy, and manpower. Large overhead is involved, and it lacks configuration flexibility. In addition, separate management tools are needed to efficiently leverage different kinds of hardware. Another solution are networking experiments in a virtualized environment. The nodes of an experiment are modeled by virtual machines (VMs) which may run any operating system, therefore latest protocol stacks may be evaluated. Communication links between the VMs may be modelled with appropriate characteristics such as rate, delay, buffer sizes, packet loss, and possibly jitter. There are a few frameworks for network experimentation in virtualized environments, e.g., Mininet [76]. They allow virtual interconnection of virtual nodes, but they do not provide detailed control over link characteristics which is essential for performance evaluation. Those tools are

not designed for the kind of task we want to achieve but more for the testing of applications in different environments where most of the time link characteristics are not important. In this paper we describe a platform for VIrtual Testbed Orchestration (VITO) that supports automation of networking experiments on a single server for the purpose of performance evaluation. A major contribution is the automation framework and the configuration of virtual links using the Linux tool TC.

3.1.2 Testbed orchestrator

Figure 3.1 gives an overview of the experimentation methodology. Physical nodes like end systems or routers are modelled by VMs and connected via one or more IEEE 802.1d [77] software bridges through VIF[2] We call those VMs NVMs[3]and the bridges EBs[4]Possibly, additional AVMs[5] may be used in the experiment to act as routers or switches. NVMs and AVMs are jointly denoted as EVMs[6]A single MVM[7]is connected to all EVMs in the testbed via a MB[8]The MVM orchestrates the virtual testbed consisting of EBs and EVMs, controls experiments, records the results and provides them for download, and finally removes the virtual testbed.

The experimentation platform itself has only the MVM and the MB running on top of a normal Linux as hypervisor system. The experimenter requires only access to the MVM but not to the host machine itself, i.e., he can create a virtualized testbed and perform experiments without root permissions on the host. Furthermore, he has a single point for the collection of experimental results. An XML-based file is used for experiment description and control. The file comprises the configuration of all EVMs and EBs, the configuration of the network, and commands that need to be issued at specified times on the VMs. First, the

[2]Virtual Interface
[3]Node VMs
[4]Experiment Bridges
[5]Auxiliary VMs
[6]Experiment VMs
[7]Management VM
[8]Management Bridge

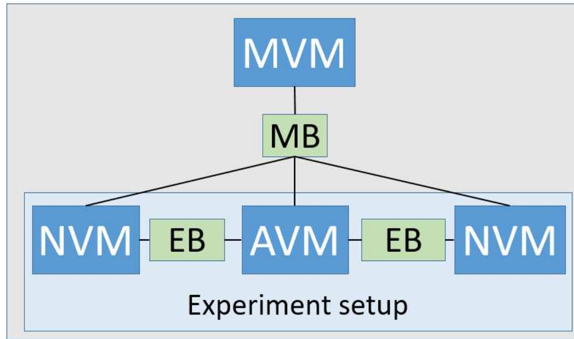


Figure 3.1: *Experiments are executed with node and auxiliary VMs (NVMs, AVMs) that are interconnected by experiment bridges (EBs). NVMs and AVMs are denoted as experiment VMs (EVMs). A managing VM (MVM) is connected via a single managing bridge (MB) to EVMs.*

MVM efficiently creates the testbed VMs using libvirt leveraging an XML-based template for VM creation. The CPU pinning option in the VM configuration may be activated to ensure that the VM is run on one CPU thread exclusively. This is helpful for EVMs running CPU-intense computations to ensure they have sufficient resources. Every EVM is equipped with a hard disk that is a snapshot of a template VM. The template foresees a single uplink interface per EVM which is assigned a random MAC address after creation. dnsmasq [78] assigns them IP addresses and makes them reachable via their host names which are configured in the description. As a result, the MVM can connect the EVMs via SSH and execute specified commands. The MVM creates required EBs, further interfaces on the EVMs with randomly generated MAC addresses, assigns them IP addresses, and connects them to the corresponding EBs. udev [79] rules are installed to rename interfaces according the description. Firewall rules are installed to prevent the usage of the uplink interface for other purposes than communication with the MVM. For the purpose of a centralized data management, a Network File System

(NFS [80]) server is installed on the MVM. This allows all EVMs to write data to a shared directory.

After creation and configuration of all EVMs and EBs, the experiment is started by executing experimentation commands specified in the experiment description. These may include comprehensive log operations, e.g., tcpflow [81] may log TCP state variables during experimentation and tcpdump [82] may log network activity on all non-uplink interfaces of EVMs. They need to be started prior to the actual experimentation and application log files may be moved to the NFS mount after completion. The actual experiment consists of a set of commands that are supposed to be executed at specified time instants after experiment start. For instance, a server process may be started on one EVM at the beginning of the experiment and clients are started slightly later on other EVMs. Commands may also change EVM configuration during the experiment, e.g., for modifying link bandwidths or simulating link failures. The virtual testbed is deleted after each experiment to avoid undesired side effects on future experiments. After deletion of the testbed, the MVN still has access to the entire data collection on the NFS mount, compresses the data, and provides it for download as a zipped tar-file.

Copying the hard disk consumes the major time fraction of EVM generation and deletion. To minimize this overhead, we use qcow2 [83] as an overlay disk image for VMs which is supported by qemu. With qcow2, all blocks initially refer to a read-only base image. If a block is written, a modified copy is stored in a qcow file. During an experimentation, only little data on the disk image is modified, in particular as logs are stored on the NFS mount. Thereby, qcow2 is very efficient and saves lots of copying overhead. As a result, the whole process of creating and deleting a virtual testbed takes less than a minute.

So far, we have only considered the use of virtual interfaces. However, physical network interface cards (NICs) may provide special optimizations like TCP segmentation offloading (TSO) [84]. With VT-d [22], PCI devices can be passed-through from the host to the VM. Therefore, it may be used to pass-through a physical NIC to a VM. VT-c [24] comprises Virtual Machine Device Queues

(VMDq) [25] which enables multiple queues per NIC, i.e., a single physical NIC (physical function, PF) is virtualized into multiple virtual NICs (virtual functions, VFs). VT-d in conjunction with Single Root I/O Virtualization (SR-IOV) [26], the virtual NICs can be passed-through to VMs. To that end, hardware-dependent kernel parameters need to be set to enable all required features. In addition, libvirt has to be configured to use a special pass-through method. With physical or virtual NICs, a VM's interface is connected to the NIC device instead of the software switch.

3.1.3 Link modelling

The Linux tool TC [85] offers access to at tool called netem (network emulator) [86] to modify the characteristics of an interface. Constant or variable delay may be added or packets may be randomly dropped or duplicated. Another mechanism offered by TC is called TBF[9] [87] which resembles a token bucket filter. It shapes a traffic stream according to a token bucket. The token bucket is configured with a rate, a burst size, and a latency. The tbf generates tokens at the configured rate, saves them up to its configured burst size, and forwards packets if sufficient tokens are available. To that end, packets need to be queued and the latency parameter determines the maximum queue size in time. As an alternative, the queue size can also be configured in bytes. If the number of tokens suffice, several packets can be forwarded at once. With tbf, rate control can be implemented for egress traffic on an interface. The burst size is usually configured as a small multiple of a maximum transfer unit (MTU). We apply first netem and then tbf to all non-uplink interfaces of EVMs with parameters defined in the experiment description. With netem, constant or variable delay may be added to individual packets and random packet drops, reordering, or duplication can be realized. With tbf, a maximum data rate of the link can be enforced by delaying and dropping packets while respecting a configurable buffer size. Also active queue management (AQM) algorithms like random early detection (RED) [88] can be

[9]Token Bucket Filter

modelled with tbf. The two tools netem and tbf can be applied both to a single outgoing interface, but the properties of the resulting stream depend the application order. In the following we use netem only for adding constant packet delay so that application order is expected to be irrelevant. We demonstrate plausible results for application order netem/tbf and show that application order tbf/netem does not work properly. As a workaround we propose the introduction of an auxiliary node so that netem/tbf and tbf/netem can be applied to a traffic stream on consecutive outgoing links which again yields plausible results. Finally, we experimentally show that the addition of an auxiliary node hardly impacts achievable throughput.

We investigate the impact of the application order of netem and tbf on a single link. A web client (curl) [89] on one NVM communicates via a vIF and a software bridge with a web server (busybox httpd) [90] on another NVM with a vIF. The guest OS uses the Linux Kernel 4.8 with TCP Cubic. The client downloads a 100 MB file from the web server via HTTP/TCP. TCP's state variables are logged with tcpflow and the packet stream is monitored with tcpdump for analysis. In a first experiment, the following two commands are applied to interface `eth1` to configure it with external parameters that are passed via the `%s` placeholders:

```
tc qdisc add dev eth1 root handle 1:0 \
    netem delay %sms %s corruption %s loss \
    %s reordering %s
tc qdisc add dev eth1 parent 1:1 \
    handle 10: tbf rate %sMbit burst %s \
    latency %sms
```

The netem command effects that the packet stream is modified with delay (ms), jitter (%), corruption (%), loss (%), and reordering (%). In our experiments, only the delay is set to 100 ms and all other values are zero. The tbf command effects that the packet stream is spaced according to a token bucket with configured rate (Mb/s), burst size (bytes), and latency (ms). This roughly models a transmission link with the specified rate and a buffer size of $rate \cdot latency/1000$. In the following, we set $rate = 50$ Mb/s, $burst = 4542$ bytes, and $latency = 50$ ms.

The chosen burst rate will be explained in detail in a later section. In a second experiment, the application order of netem and tbf is interchanged. We apply the same configuration for transmission from the client to the server and vice-versa. Table 3.1 summarizes the results. While download time and goodput are rather

Table 3.1: *Performance metrics for a download of a 100 MB file with netem/tbf and tbf/netem configured on a single link with bandwidth of 50 Mb/s.*

tool order	download time	goodput (Mb/s)	avg. CWND (MSS)	avg. RTT (ms)
netem/tbf	19.7s	40.6	821	231
tbf/netem	19.3s	41.5	2013	427

similar for both application orders, packet loss, avg. TCP congestion window size (CWND), and avg. roundtrip time (RTT) differ significantly. For further analysis, we consider CWND and RTT over time which is illustrated in Figure 3.2. With netem/tbf, the RTT varies between 200 ms and 250 ms, and the CWND between 500 and 1050. In particular, CWND and RTT decrease after the occurrence of a lost packet. This is different with tbf/netem as no packet is lost. As a result, the CWND increases up to a maximum value of 2300 MSS and the RTT oscillates between 420 and 520 ms. RTTs in this order of magnitude are unexpected because the configured transmission and queueing delays can range only between 200 ms and 300 ms. Thus, the application order tbf/netem causes undesired behavior and should be avoided for experimentation.

We now interconnect the client and the server NVM via another AVM and software bridges. The configuration is illustrated in Figure 3.3 for the application order tbf/netem. Thus, netem/tbf or tbf/netem are applied to consecutive links instead to a single one. The client and server NVM see only single links with combined properties. Table 3.2 reports the experimentation results which hardly differ from the experiment with netem/tbf. The same holds for CWND and RTT over time for which figures are omitted. Thus, netem/tbf may be applied together on a single outgoing interface. If tbf/netem is the desired application order, an

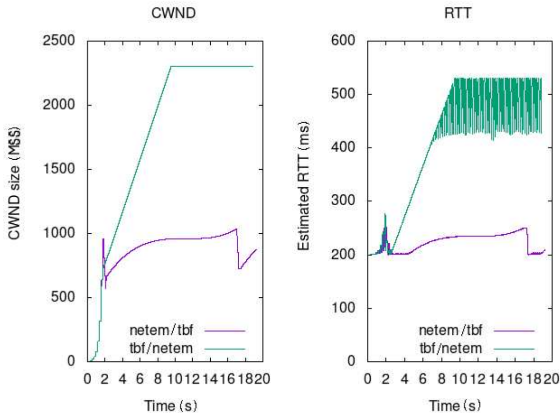


Figure 3.2: *TCP's congestion window and roundtrip time for a single TCP flow on a link with 100 ms delay, a bandwidth of 50 Mb/s, and a buffer with a maximum latency of 50 ms; tbf's burst size is configured with 4500 bytes.*

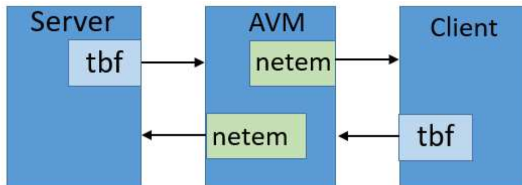


Figure 3.3: *tbf/netem is configured individually on two consecutive links through introduction of an additional EB and AVM that just forwards traffic.*

auxiliary node may be used to ensure correct behavior.

An additional AVM may add some delay. We show that this delay is so small, that it hardly influences experimentation results. To that end, we perform similar

Table 3.2: *Performance metrics for a download of a 100 MB file with netem/tbf and tbf/netem individually configured on two consecutive links.*

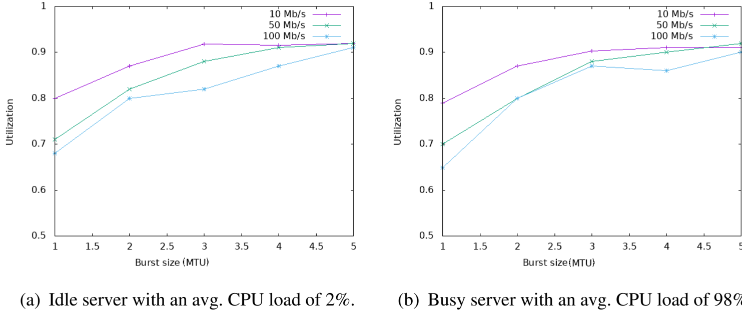
application order	download time	goodput (Mb/s)	avg. CWND (MSS)	avg. RTT (ms)
netem/tbf	19.1s	41.8	834	224
tbf/netem	18.9s	42.3	845	229

experiments like above but deactivate rate control and add a delay of 0 ms and 1 ms, respectively. We perform the experiments 20 times. Table 3.3 summarizes the utilization. Even without any base delay, the presence of the AVM is hardly visible by the increased download time and with a base delay of 1 ms, the download time is almost the same. Thus, AVMs add so little delay that it cannot even be perceived for small positive based delay. To find a reasonalbe burst size for

Table 3.3: *Download time for various configurations depending on one-way delay with a configured bandwidth of 100 Mb/s and a 100 MB file size.*

delay	w/o AVM	w/ single AVM	w/ jLISP
0 ms	8.68s	8.70s	9.21s
1 ms	8.72s	8.73s	9.22s

tbf we first think of a simple rate controller that ensures a maximum bit rate of C. After transmission of a packet with size B, it transmits the next packet not earlier than after $\frac{B}{C}$ time. If the machine performs that task only slightly late, this reduces the maximum achievable data rate. To cope with the problem of late transmission, tbf uses a token bucket description for spacing. tbf continuously generates tokens and saves them in a bucket which is limited by its burst size. Packets are queued for transmission. A packet is sent if the number of tokens in the bucket is at least the packet size. If the number of tokens does not suffice, the machine retries again when enough additional tokens have arrived. On the one hand, this mechanism assures that transmission capacity is not lost if a packet is



(a) Idle server with an avg. CPU load of 2%.

(b) Busy server with an avg. CPU load of 98%.

Figure 3.4: *Impact of configured burst sizes on utilization depending on configured bandwidths.*

sent slightly later than possible, on the other hand it allows transmission of multiple packets so that packet bursts may be transmitted. Therefore, the burst size should be set only large enough that the full transmission capacity can be leveraged, but also as little as possible to keep bursts small as the intention of a spacer is a smooth traffic stream. In the following we evaluate the required burst size for various bandwidths. Figure 3.4(a) compiles the utilization for various burst sizes for a server NVM with little CPU load. The figure shows that large bandwidths require large burst sizes to minimize download times by fully leveraging the configured bandwidth. We perform the same experiment with a server NVM that performs other tasks in parallel so that its CPU load is close to 100%. The results in Figure 3.4(b) show that almost the same download time values are achieved because TC is granted high priority as it is running in kernel. Both experiments were repeated 100 times which resulted in a confidence interval of less than one percent in each direction for an alpha value of 0.95. We derive from Figure 3.4(b) recommendations about minimum burst sizes for specific values of configured bandwidths that are needed so that the full transmission speed can be achieved with tbf. The results are summarized below The values may depend on software

and hardware.

bandwidth (Mb/s)	burst size (MTU)
<= 10	3
<= 50	4
<= 100	5
> 100	7

3.1.4 CI/CD

When using VITO, one of the main goals is simplicity and the ability to conduct large experiments with as little work as possible. A step further would be using a CICD[10] system like Jenkins [91] or Gitlab CI [92] to automate the whole experiment process. A docker container for the CICD runner will read the experiment definitions from a git repository, copy them to the test system and starts the experiments. A second Job that will be started after the experiments finished could then be used to push the results back into a repository or to a transfer storage for further processing. Doing both in the same pipeline is not recommended due to the variable length of experiments. Long running experiments lead to overhead and blocking slots for other runners while doing nothing and only waiting for experiments to finish.

3.2 IP

This chapter is based on the paper "A Semi-Virtualized Testbed Cluster with a Centralized Server for Networking Education" published at ITC 30
Practical courses are an important part of networking education. Students learn to configure devices and interconnect them with cables and switches. Such courses are traditionally based on physical testbeds consisting of PCs, routers, and switches. While offering real hands-on experience, this approach has the drawback that it requires lots of space, energy, and maintenance effort. With

[10] Continous Integration / Continous Deployment

progress in virtualization technology, fully virtual testbeds emerged and were used in some networking courses. In these testbeds, PCs, routers, and switches run as virtual machines (VM) on a server, thereby avoiding some shortcomings of physical testbeds. However, fully virtual testbeds do not provide hands-on experience which is an important learning target and fun factor of practical networking courses. We believe that hands-on experience is an important part of networking labs. We observe students having problems realizing limitations and problems with physical cabling regarding available ports, cables etc. as well as problems organizing their cabeling work. Therefore it is important to give them the opportunity to use real hardware as part of the learning experience. In this paper we focus on lab systems that allow hands-on experience. We distinguish between three different types of architectures depicted in Figure 3.5. Traditionally, a student workspace consists of a physical testbed that allows to interconnect several computers and routers. An entire lab system is composed of several such testbeds and an additional server to provide IS[11]. As an alternative to a physical testbed, a semi-virtualized testbed with a dedicated server per student workspace may be used. Computers and routers run as VMs on the dedicated server and a patch panel allows to interconnect their interfaces with cables. The testbed consists of a PW[12] and a VW[13] with the VW running on the virtualization server. Due to the PW, the testbed is only semi-virtualized. This preserves all benefits of physical testbeds. In this work, We present a semi-virtualized testbed cluster with a central server for multiple student workspaces. The central server hosts the IS and the VWs for multiple testbeds that are mapped to different PWs.

At the University of Tuebingen we offer practical networking courses since 2004. Initially, our curriculum was based on the concept of Liebeherr and Zarki [93] using physical testbeds. In 2012, we reworked the lab content and substituted the physical testbed by semi-virtualized testbeds with dedicated servers [94, 95]. Recently, we further elaborated that approach towards a testbed

[11] Infrastructure Services
[12] Physical Workspace
[13] Virtual Workspace

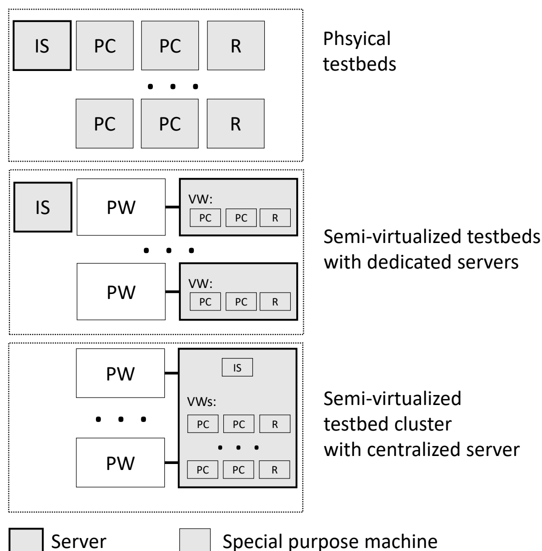


Figure 3.5: *Three types of lab systems for networking education with hands-on experience.*

cluster with a centralized server.

Virtual testbeds can be easily managed with the help of an orchestration framework. Such frameworks are widely used and a common technology today. In contrast, orchestrators for semi-virtualized testbeds which fit the requirements for practical networking courses are hardly found. A reason for that is that orchestrators for semi-virtualized testbeds are not trivial to implement as they depend on the actual hardware platform and components used in the testbed, so in most cases a fit for purpose solution needs to be built and preexisting frameworks are not of much use at all. Moreover, they need to map virtual components to physical hardware.

In this work we present an architecture for a semi-virtualized testbed cluster with a central server offering multiple student workspaces. It facilitates automatic orchestration and simplified management and introduces multiple new features that are enabled through the new architecture. The single server is based on the x86 platform and makes use of current virtualization techniques such as VT-x [18], VT-d [22], VT-c [24], and SR-IOV [26]. They enable VM performance close to physical machines, in particular regarding network throughput. For orchestration purposes, I developed the “Lab Orchestrator for Semi-virtualized Testbeds” (LOST). It is a Python-based platform for orchestration of VMs in a semi-virtualized environment. It makes use of the KVM [27] hypervisor to run VMs and leverages features of libvirt [9] to manage them.

LOST groups several student VMs into virtual workspaces and maps them to physical workspaces. The physical workspaces enable the students to interact with the VMs in a similar way as with physical machines and configure network topologies with the help of cables and switches. In addition, LOST supports usage of USB devices plugged into physical workspaces by associating them with student VMs.

LOST supports different types of VMs with different roles (clients, servers, and routers). It instantiates them from templates that are derived from a base image. To improve the manageability and to speed up orchestration of VMs, we developed a layered concept for file system access of VM images: a jointly used base image and additional layers for typing and individualization reduce memory copies upon instantiation of VMs.

3.2.1 Concept

In this section, we present the overall architecture for the semi-virtualized student testbed cluster. A single lab server hosts multiple virtual workspaces that are mapped to physical workspaces for physical user access. Additional infrastructure VMs are used to provide services for the virtual workspaces and the students. We suggest an optimized, layered storage organization for VMs to reduce man-

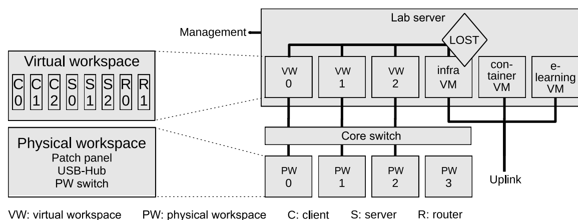


Figure 3.6: *Architecture of the semi-virtualized lab infrastructure on a single server.*

agement overhead and to improve software consistency. The technical realization of the mapping is described in Section 3.2.3 as it depends on the hardware platform of the lab server. Finally, we compare the new concept for semi-virtualized student testbed cluster with other approaches regarding acquisition cost, energy consumption, and maintenance effort.

3.2.1.1 Architecture of the Semi-Virtualized Testbed

Figure 3.6 shows the overall architecture of the semi-virtualized testbed. A single lab server hosts virtual workspaces (VWs) with SVMs[14] as well as IVM[15]. A physical workspace (PW) consists of a set of devices giving access to the SVMs of a virtual workspace. The “Lab Orchestrator for Semi-virtualized lab Testbeds” (LOST), presented in detail in Section 3.2.2, is a collection of scripts that run on the host and on IVMs. It sets up the virtual workspaces and maps them through a core switch to the physical workspaces so that students can interact with SVMs and interconnect them with cables and unmanaged switches.

A *virtual workspace* is a set of SVMs that represent a students’ workspace on the server. Figure 3.6 illustrates a typical setup which consists of three client (C) SVMs, three server (S) SVMs, and two router (R) SVMs. All virtual workspaces

[14] Student VMs
[15] Infrastructure VM

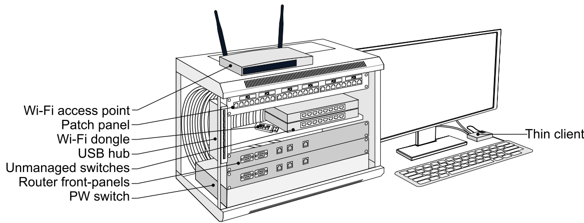


Figure 3.7: *The physical workspace provides access to the desktop and outlets of the virtual workspace.*

use the same setup which is defined by a template for LOST.

A *physical workspace* is the actual workspace that allows students to interact with the VMs. Figure 3.7 depicts a physical workspace. It consists of a 19 inch cabinet, a thin client for the student user interface, and I/O devices. The cabinet contains the following components. The PW switch is a managed switch at the bottom of the cabinet. It connects the cabinet via the core switch to the lab server hosting the VMs like shown in Figure 3.9.

The PW switch demultiplexes the interfaces of the VMs in the corresponding virtual workspace to individual switch ports. The ports for the network interfaces of clients and server VMs are connected to a patch panel at the top-most position in the cabinet. The ports for interfaces of the routers are connected to custom-designed front-panels that look like traditional routers (see Figure 3.7). To enable the students to set up more complex network topologies, two additional unmanaged switches are placed on a compartment sheet in the middle position of the cabinet. For experiments involving wireless technology based on IEEE 802.11, a Wi-Fi access point is placed on the top of the cabinet and USB Wi-Fi dongles for selected VMs are provided. These dongles can be plugged into a USB hub that is mounted on the left side of the cabinet and connected to the server.

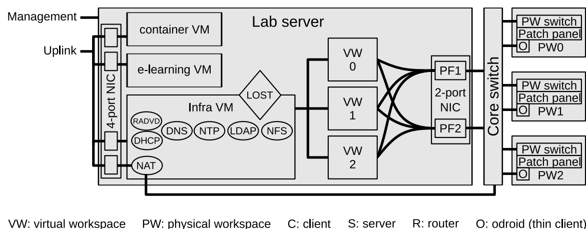


Figure 3.8: *Overview of components, services, and connections of the single server testbed.*

3.2.1.2 Interconnection Network and VW-PW Mapping

Figure 3.8 gives an overview of the internal structure of the single server lab infrastructure and the connection between the different components. In the figure, the server hosts three VWs and each VW holds three VMs. Each SVM of a VW contains one INI[16] and at least one student network interface (SNI) which can be configured by the students. The number of SNIs depends on the role of the SVM. All INIs are connected through an internal bridge per VW on the host to an IVM that provides infrastructure support (see Section 3.2.1.3). The patch panel of a PW contains outlets for the SNIs so that they are physically accessible and the students can perform the cabling for the labs. To that end, all SNIs are realized as virtual interfaces on a two-port NIC that is connected to the core switch. The mapping of virtual interface to physical interface depends on the absolute numerical identifier of the virtual interface. All even numbers are mapped to the first port of the two-port NIC and all odd numbers are mapped to the second port of the two-port NIC.

Figure 3.9 illustrates the mapping of the SNIs to the patch panel and the connection of the thin clients to the lab server. To distinguish the different interfaces, each is placed in a unique VLAN [96] with the following tag scheme: `$pw$svm_num$interface_num`. That means, all VLAN tags consist of

[16]Infrastructure Network Interface

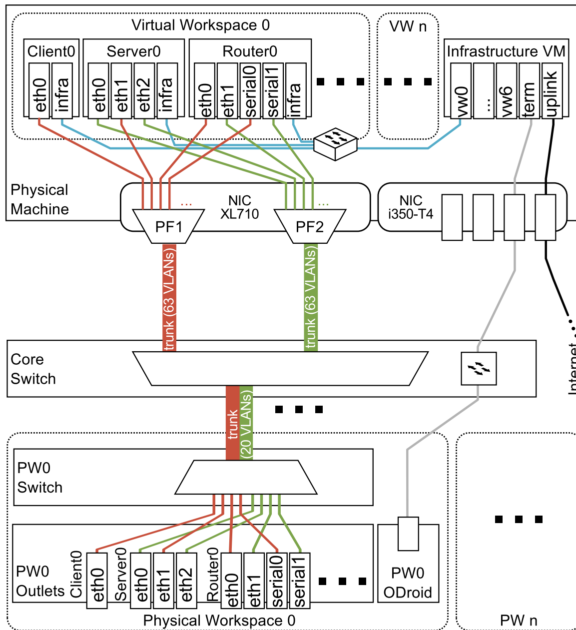


Figure 3.9: *Interconnection of student VMs on the lab server with patch panels on PWs.*

a three-digit number with the most significant digit specifying the physical workspace, the second-most significant digit specifying the SVM, and the least significant digit specifying the interface within an SVM. VLAN tags are automatically added/stripped at the transition from SVM to host. The mapping of `$pw` to actual PW is dynamically configured by LOST (see Section 3.2.2).

The host forwards incoming packets to the corresponding SVM according to the VLAN tag. In the other direction, packets from the SVM are forwarded to the core switch as two big VLAN trunks via the two-port NIC.

On the core switch, the VLAN trunks are demultiplexed into sub-trunks that

are forwarded to the PWs specified in the VLAN tag. Within a PW the VLAN trunk is processed by a managed switch. The switch is used to re-order the VLANs to the corresponding port in the patch panel and to strip the VLAN tags. This mechanism ensures that the students do not see any VLAN tags.

3.2.1.3 Infrastructure Support

We run three supplementary IVMs on the lab server that host the lab platform and provide services. The first IVM facilitates basic network services for both the SVMs and the thin clients in the PWs. Each VW gets its own dedicated subnet whereas all thin clients are placed in the same subnet. To that end, a DHCP server and a router advertisement daemon distribute IPv4 and IPv6 addresses, routes, and information about additional services like DNS and NTP. The DNS server uses the following canonical naming scheme for names in the lab: `${host_name}.tb${vw_num}.inetlab`. Queries to resources in the Internet are forwarded to an external DNS server. The local NTP server syncs its time information with an external time server and ensures that the time in all VWs is in sync. This IVM also acts as NAT gateway so that the thin clients of the physical workspaces can connect to the Internet. The first IVM also includes a central LDAP [97] directory and an NFS [98] server. The LDAP directory stores the accounts for the students as well as their user rights within the SVMs. An NFS server holds the home directories of the students as well as some initial configuration data and scripts that the students use during the exercises. The other two IVMs host the e-learning platform for the course and provide LXC [17] containers for special home exercises, respectively.

3.2.1.4 Simplified VM Maintenance

We first explain the need for simplified VM maintenance in the context of testbeds, then we review concepts related to our solution, and finally we introduce the new VM maintenance method.

Motivation In our testbed, many VMs need to be maintained, i.e., configuration changes and software updates need to be applied. Manual maintenance is a lot of effort and error-prone. Therefore, maintenance work is often automated with scripts or configuration management tools like puppet [99]. Different maintenance times or additional changes of individual VMs by the administrator lead to diverging VM images. However, different VM images are undesirable in testbeds where at least all VMs of the same type should be identical at the beginning of an exercise.

Related Concepts The storage image solution QCOW2 is an updated copy-on-write hard disk container to provide hard disk storage for Qemu-based VMs. It allows to save a VM image as a stable base image file and to store later changes to the VM image in a snapshot file. This facilitates the reset of the VM by deleting the snapshot file.

In [2], this technique has been used to share a major portion of a VM image on hard disk among multiple VMs. They are booted from the same QCOW2 base image and record their image changes in individual snapshot files. Various base images supported different VM types. This technique was used in [2] to save disk space and in particular to reduce copy operations for VM setup.

Stacking file systems can be done with OverlayFS for Unix/Linux . It combines a lower and upper file system, i.e., the lower file system serves as a stable base and the upper file system accounts for differences. Thus, the upper file system tracks changes, i.e., file generation, deletion, and modification. File requests are served from both combined file systems like from a single file system. OverlayFS can be applied recursively, i.e., the lower file system may be another OverlayFS file system.

Multi-Layer VM Images We present a novel maintenance method for VMs with similar configuration. It is based on the observation that the VMs share a major portion of their images and the differences result from a moderate number of additional configuration actions. It is helpful for maintaining VMs with differ-

ent host names, network configuration, and services. In [2], several base images are needed to support different VM types. Now, the objective is to utilize only a single base image for different VM types.

We diversify VMs by leveraging OverlayFS and providing configuration changes in the upper file system of OverlayFS. To simplify the provisioning of the upper layer file system, We define templates for SVMs with root file systems for clients, servers, and routers that are further adapted to the specifics of individual VMs.

However, OverlayFS becomes active only after loading of the operating system kernel has completed. Until then, VMs write all data to the lower file system. If multiple VMs leverage the same lower file system, inconsistencies will occur. To avoid inconsistencies due to writes from different VMs to the same base image, an additional QCOW2 is utilized as protection layer. As a result, VMs are booted from a joint base image containing the lower file system and modifications are tracked in individual QCOW2 snapshots. When OverlayFS is started, VMs are diversified through the configuration contained in the upper file system, the adaption layer. This leads to a setup where a VM references a QCOW2 image as storage which is also a reference to the another storage image, namely the mentioned base image. For the VM this looks like a single hard disk even though it is a stack of two copy on write file systems. On top of that the OverlayFS is managed by the OS which is aware of this layer.

When student work on VMs, they apply configuration changes. It is desirable to easily undo them and restore the VM to a defined starting point. To that end, We implement another OverlayFS layer above the VM diversification to track all further modifications to the VM in a second upper file system, the user layer.

Technically, the base image combined with the secure base appear one disk, the adaption layer appears as another disk, and the user layer appears as a third disk. These three disks are input to OverlayFS whereby their order matters. However, the disks do not necessarily always appear in the same order, which affects their numbering. Thus, the disk numbers cannot be used as input for OverlayFS. We fix that problem with the following workaround. The disk contains metadata

including a disk label as another identifier. This disk label is set after creation of the disk and remains stable. Therefore, we specify the inputs for OverlayFS using the disk labels rather than the disk numbers. In our prototype, We use ext4 as file system type and e2label [100] as disk label implementation.

Figure 3.10 illustrates the proposed concept. A single LVM[17] [101]) volume constitutes the base image jointly used by all VMs. The protection layer is individual for all VMs and achieved through QCOW2. Its snapshot (secure base) intercepts initial runtime modification of the VM to protect the joint base image. The adaption layer is also individual for all VMs and implemented through OverlayFS. It combines a VM's base protection layer with the initial upper file system holding the VM's configuration data and minor runtime modifications. Finally, the user layer is also implemented through OverlayFS and holds all student changes in its upper file system.

After completion of an exercise, the user layers can be deleted to reset VMs to the states defined by their adaption layers. To maintain all VMs in the testbed, only the base image needs to be updated provided that maintenance operations do not affect the folders with the configuration data of the adaption layer. Therefore, this concept provides high flexibility, minimizes maintenance overhead, and reduces storage requirements.

3.2.1.5 Comparison: Acquisition Cost, Maintenance Effort, and Energy Consumption

In this section I provide a comparison regarding acquisition cost, maintenance effort, and energy consumption of the following three lab system approaches: 1) a physical testbed, 2) a semi-virtualized testbed with a single server per testbed, 3) a semi-virtualized testbed cluster as proposed in this work.

[17] logical volume manager

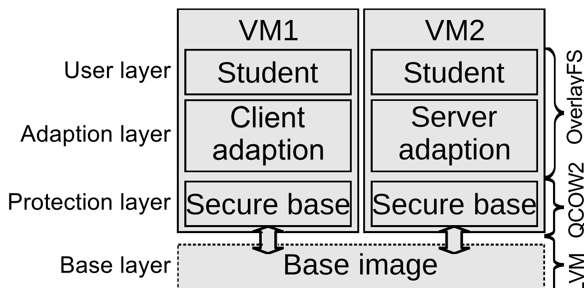


Figure 3.10: *File system access based on layered storage.*

3.2.1.6 Acquisition Costs

We estimate acquisition costs for the components of the three different lab system approaches. I assume reasonable, cost-efficient prices as performance is not crucial.

For the physical testbed We assume that cheap all-in-one PCs are used and upgraded with a 4-port network card. This will result in around 200 Euro per PC and 20 Euro for the network card. In addition, two routers are needed. Recent or high-performance router models are not needed since all basic functions used by students are implemented in router firmware for more than 20 years. Therefore, cheap, old models may be utilized which are available for 50 Euro each. In total a single testbed costs 1440 Euro. An infrastructure server providing services can be acquired for around 1200 Euro in total. A typical lab system consisting of 6 testbeds cost roughly 9500 Euro.

The semi-virtualized testbed with a dedicated server needs a physical workspace which costs around 800 Euro each, including the PW switch. In addition, a dedicated server per PW is needed as host machine to run the hypervisor for the VMs. We use the same machine type as for the infrastructure server for

this purpose. A lab system with 6 work-spaces amounts $7 \cdot 1200$ Euro $+ 6 \cdot 800$ Euro $= 13200$ Euro.

The proposed architecture only needs one big server for the entire testbed cluster which costs around 7000 Euro. The physical workspaces are the same as for the semi-virtualized testbed with dedicated server and also cost 800 Euro. In addition, a core switch is needed which costs around 400 Euro. This results in total costs of 7000 Euro $+ 6 \cdot 800$ Euro $+ 400$ Euro $= 12300$ Euro.

Maintenance Effort In the past, We encountered a maintenance effort of about 1 hour per physical machine and semester in all three lab system approaches. That means, for the physical testbed, each server, PC, and router requires that maintenance effort. For the semi-virtualized testbeds, the servers require that maintenance effort, but in addition, the VM images need to be kept up to date. In case of one server per PW, care needs to be taken for the VMs on all servers. Experience has shown that the required effort scales with the number of PWs although there are options for automation. In practice, We needed about 3 hours per PW and semester for maintenance. In case of a single server for all PWs, only a single VM needs to be maintained. That requires only 3 hours maintenance effort per testbed cluster and semester.

Energy Consumption The power consumption of small all-in-one PCs is estimated with around 20 W. We observe power consumption of 200 W for small servers and 300 W for a big server. The PWs are powered with 25 W each for all active components (PW Switch, small switches, USB).

The following table compares the three approaches in terms of cost, maintenance and power consumption.

Testbed type	Acquisition cost (Euro)	Maintenance effort (h)	Power consumption (W)
physical testbed	9500	49	1160
semi-virtualized testbed	13200	18	1550
semi-virtualized testbed cluster	12300	3	450

3.2.2 A Lab Orchestrator for Semi-Virtualized Testbed clusters (LOST)

As no existing orchestration platform like OpenStack provides the features that are required for our use case, We developed *Lab Orchestrator for Semi-virtualized lab Testbed clusters* (LOST) as a new one based on the same principles as VITO described in the last chapter. It is especially designed to support the VM storage concept, the simplified VM maintenance, and the template system for SVMs and VWs with their mapping to PWs presented in Section 3.2.1. It mainly consists of a set of Python scripts with some additional bash scripts.

LOST is configured via config files including the following parameters:

- The templates for the different SVMs include the adoption layers and specify the the hardware resources like the amount of network interfaces (NICs), CPU, and RAM. We have templates for clients, servers, and routers. E.g., the routers provide a Cisco-like CLI, the clients have a graphical desktop environment, and the servers are terminal machines that run several services.

- The composition of a VW consisting of different SVMs is defined. In our use case, We have three clients, three servers, and two routers (like depicted in Figure 3.6) as that fits best to the different exercises in the courses. It is easily possible to define other composition, the available hardware resources on the lab server are the only limitation.

- The amount of available PWs is configured, so that a VW for every PW can later be instantiated. We have 6 workspaces.

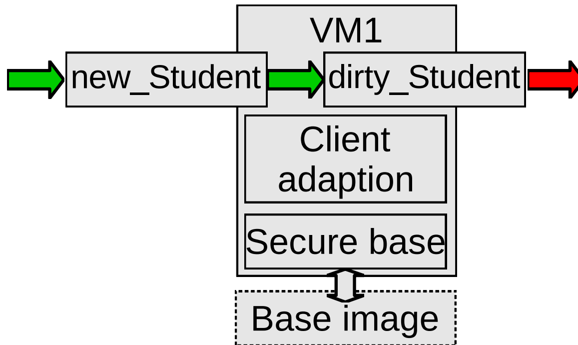


Figure 3.11: *Wiping the user layer restores the original SVM states*

- A default mapping of VW to PW is provided. Figure 3.12 depicts such a default mapping.

With the configuration files as input, LOST instantiates the different VMs and configure the core switch as well as the PW switches according to the config files. E.g. the different VLANs are configured. The configuration of the switches leverages software-defined networking (SDN) technology. Generally, LOST supports different southbound interfaces like SNMP [102], OpenFlow [54] or NETCONF [55]. For compatibility reasons with as many switches as possible, We currently selected SNMP.

The instantiation process includes the initialization of the protection layer and the adoption layer (see Section 3.2.1.4) for the individual VMs and assigning hardware resources such as NICs, CPU, and RAM to them. LOST also manages the VM state, i.e., this includes starting and stopping of VWs, and monitors the different VMs. VMs can either be managed individually or as bulks. After a lab day or in case of a heavily misconfigured SVM, LOST can wipe the student layer so that the students can start over again with a well-defined VM states (see Figure 3.11).

In case of a hardware failure on a PW, it is possible to move a VW to another

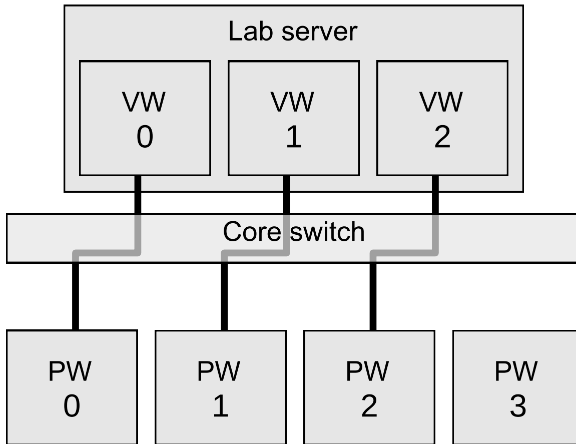


Figure 3.12: *Failre-free condition*

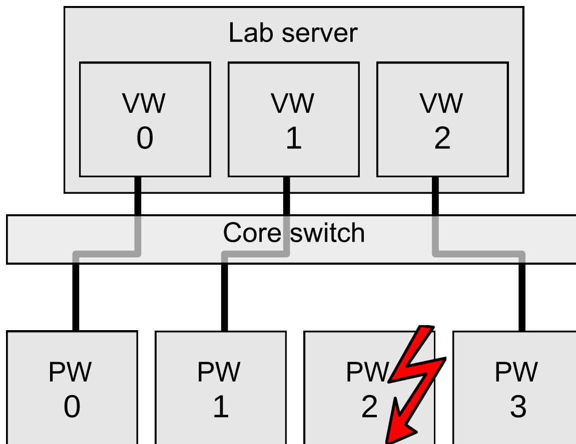


Figure 3.13: *Re-mapping in case of failed PW.*

PW as depicted in Figure 3.13. To that end, LOST reconfigures the affected PW switches and the core switch. The advantage of this mechanism is that students only have to re-apply the cabling on the new PW, but all their configuration and data of the VMs are available at the new location.

3.2.3 Implementation

In the following, We describe the hardware, software, and virtualization platform that I used to implement the semi-virtualized testbed.

3.2.3.1 Hardware Platform

First, We outline the components included in the single lab server and the additional equipment used to provide the physical interaction with the VMs.

Server We use a DELL PowerEdge T430 based on the current Intel server platform as base system for the lab server. The server is equipped with two Intel Xeon CPU E5-2660 v4, 128 GB RAM, and a hardware RAID with level 6. Figure 3.9 shows that two additional Intel network cards provide the network interfaces for the SVMs and IVMs. An Intel XL710 40GbE SFP+ two-port NIC which supports 64 virtual interfaces per physical interface, relays the network interfaces for the student VMs. An additional Intel i350-T4 NIC provides the uplink to the Internet for the three IVMs and connects one of them to the lab network. The onboard NIC of the server is used as management access for the server.

As we want to be able to equip the student VMs with USB devices like Wi-Fi dongles or headsets, a USB3 hub is attached to the server. It connects the USB hub in each PW via a fast port USB to the lab server. The hub allows to attach USB devices like Wi-Fi dongles or headsets to the SVMs.

Additional equipment The core switch connecting the different PWs to the server is a Netgear S3300-28X ProSAFE 24-Port Gigabit. It includes two SFP+ 10G uplink ports which are connected to the Intel XL710 at the lab server. Each

physical workspace has two connections to the switch: one is used for the thin clients and the other is used to connect the cabinets with the server. Each cabinet contains a managed HP/Aruba switch as PW switch to de-/multiplex the testbed uplink to different physical ports on the switch. They are connected to a patch panel containing keystones for the network outlets of the VMs. The USB hub in each cabinet is connected to the USB3 hub at the lab server via an active USB cable. A Hardkernel odroid c2 is used as thin client. This platform provides enough computational power and RAM to act as graphical client or terminal client for 8 VMs and to run the Chrome web browser. The odroids are connected to a 24 inch full-hd (1080p) monitor, keyboard, and mouse for interaction.

3.2.3.2 Software Platform

We briefly outline the software used on the host and the different VMs.

Host The server itself only acts as host for VMs. For stability purposes, more efficient use of resources, and reduced security risks, the server contains a minimal environment. We chose Gentoo Linux [103] as operating system (OS) on the server as it gives full control to the administrator. With the help of USE-flags it is possible to define which parts of a certain package should be installed. This way, only the parts and features of a package that are really needed and wanted are installed. Additionally, Gentoo does not force the user to use a certain component, e.g., the user can freely select the init system. Furthermore, in our experience Gentoo is more stable compared to other Linux distributions like, e.g., Ubuntu.

We chose kvm [27] as hypervisor because it is already part of the Linux kernel. As kvm cannot be used directly, a supplementary virtualization tool like qemu [28] is required. Qemu makes use of kvm and provides additional virtual devices by itself. Also parts of LOST (see Section 3.2.2) are run on the host. To ease the handling of the VMs in LOST, We designed LOST to utilize the libvirt [9] framework which already implements functions to create, start, stop, and delete VMs. These operations are extended with custom operations, like applying a template to a VM during creation as needed.

Infrastructure VMs (IVMs) As already explained in Section 3.2.1.3, we run three additional server VMs providing services for the lab and hosting the lab platform. The first IVM runs a isc-dhcp [104] DHCP server and a radvd [105] daemon to configure the network addresses. A BIND [106] DNS server is used to resolve the IP addresses for internal and external resources. The LDAP directory runs on an OpenLDAP [107] server and the NFS kernel server provides NFS directories. The reference implementation [108] from the NTP project is used as NTP server.

We use the nginx [109] webserver to host an instance of the iLab e-learning platform [110]. In conjunction with shellinabox [111], nginx also enables access to LXC [17] containers. The containers are used in an introductory assignment which takes place before the practical course. This assignment compiles an introduction for the students how to work on a Linux shell, do basic network configuration and simple experiments.

Student VMs (SVMs) All SVMs are based on a single base image. The template system of LOST derives different kinds of SVMs like client or server from the base image. This base image already contains most of the tools and software for all different types of SVMs. Among this software is the LXQt [112] desktop environment used for clients, services, and daemons used for the servers and the quagga [113] routing suite from which We use vtysh (a Cisco like CLI) for virtual routers. However, the functionality of this software is disabled in the base image. The templates activate the functionality required for their use case. The thin clients can connect to the SVMs via spice [114] which transports the display and input devices of the VMs.

As an alternative to VMs, containers could be used. However, we decided to use VMs because VMs have their own networking stacks so that VM nodes are isolated against each other and students observe a similar networking performance as with real hardware. The performance obtained in this setting is sufficient for most applications in general and suffices for all exercises carried out by the students. Using VMs instead of containers provides sufficient flexibility

for future extensions. We planned to define an data center networking lab where VMs act as servers hosting containers. Implementing data center servers as containers would lead to a rather unrealistic solution. Thus, the decision to utilize VMs instead of containers on the host makes LOST future-proof.

3.2.3.3 Virtualization Platform

In the following, we describe virtualization features and techniques that we use on our infrastructure. First, we briefly outline the basic requirements for virtualization on the x86 architecture. After that, we explain the mechanisms used to pass-through PCI and USB devices from the host to the VMs.

Hardware-Assisted Virtualization for x86 The x86 platform itself is not virtualizable in hardware, which means that VMs must be emulated in software. Software-based VMs lack performance. Hardware-assisted VMs require some extensions to the x86 architecture: VT-x [18] enables basic hardware acceleration for virtualization. It includes additional instructions and registers to implement an additional privileged system and hardware-based shadowing for the memory management unit (MMU). This way, VT-x permits entering and exiting a virtual execution mode. In this mode the host OS remains protected while the VM OS perceives itself as running with full privilege. Second level address translation (SLAT) is an additional extension to the MMU which further increases the performance. It basically treats each physical address of a VM as a virtual address on the host. This prevents software lookups to determine the actual physical memory address of VM memory. SLAT is implemented on the Intel platform as extended page tables (EPT) [20]. Additionally, EPT is a requirement to start a VM directly in real mode with unrestricted access. Typically, hypervisors emulate most guest access to interrupts and the advanced programmable interrupt controller which requires the exit and entry of a VM. This is time-consuming and a major source of overhead. Advanced programmable interrupt controller virtualization (APICv) [19] eliminates lots of VM exits and can increase performance significantly.

Pass-through of PCI Devices VT-d [22] provides an IOMMU [26] which allows to pass-through devices, e.g., NICs from the host to the VMs. With the help of IOMMU groups, different devices are isolated against each other and secure memory access is ensured. Linux kernels later than 4.1 require the PCIe Access Control Services (ACS) [23] feature for separated IOMMU groups. VT-c [24] comprises Single Root I/O Virtualization (SR-IOV) [26], an extension to the PCI standard, and Virtual Machine Device Queues (VMD-q) [25]. SR-IOV classifies devices in physical functions (PFs) and virtual functions (VFs). A PF is a full-featured PCI device, e.g. a NIC or a graphics card, which can run on its own. A VF is a lightweight PCI device which cannot run independently of a PF. The VF shares some resources with the PF that manages this VF. VMD-q enables multiple hardware-based queues per NIC which are internally connected to a bridge. Together, VT-d and VT-c instantiate a dedicated hardware queue per VF which appears on the host as a virtual NIC. This NIC can be exclusively passed-through to a VM so that the host does not see the device any longer. With this mechanism the virtual NICs achieve a performance close to dedicated physical NICs. As all VFs communicate over the common physical port of their PF, VLAN tags are used to differentiate traffic from and to different VFs. To be able to distinguish traffic for the different VFs, they are typically mapped to different VLANs [96].

Pass-through of USB Devices To pass-though USB devices, the entire address of a device on the USB bus is mapped to the address space of a VM. This process can be automated with the help of special udev rules [79] that trigger qemu to map a newly plugged in USB device to a specified VM. This requires a lookup in the LOST configuration so that USB devices must be identifiable. Typically, a USB device can be identified by a unique identifier (UUID) in the device description field of its ROM. However, some hardware manufactors and vendors disregard the uniqueness of UUIDs and assign the same UUID for multiple devices or entire batches thereof. That problem pertains to the USB Wi-Fi dongles and USB headsets in our system. I developed the following workaround

to connect USB devices with equal UUIDs in VMs.

When a USB device is plugged into the USB bus, the host loads the driver for that device. If the USB device is a network device, the host looks up its unique MAC address. If that MAC address is assigned to some VM in the LOST configuration, the host unloads the USB driver for that device and passes its USB bus address through to the configured VM. This workaround allows to assign a specific Wi-Fi dongle to a VM, which is needed because some scripts and udev rules in the VM contain their MAC address. The described workaround works for Wi-Fi dongles, but not for USB headsets as they do not have MAC addresses. Fortunately, a VM does not require a specific headset, but can work with any headset connected to the VM. Therefore, the $n-th$ plugged-in headset is passed-through to a specific VM that is configured with LOST depending on n.

4 Hybrid Access

Parts of this chapter are based upon text of a future unpublished technical report. The traditional approach to connecting households or small companies to the internet is by the use of a single line, most often copper based or wireless. In remote areas, mobile networks and satellite communication are used. In situations with unreliable connections or low bandwidth this does not provide a satisfactory user experience. In such cases, it may be possible to bundle multiple access methods, such as copper and mobile networks, to offer the customer more bandwidth or better reliability. This principle is commonly called hybrid access (HA). In this section different methods for HA are compared and tested using the VITO infrastructure.

4.1 Load-balancing Concepts for Heterogenous Links

In this section, we compare the performance of MPTCP and simple PBLB, proposed in [**?**], for HA by experiments in a virtual environment using real networking stacks. Then, we propose algorithms for FBLB in Section 4.1.2. As an important feature of all these protocols, information about link bandwidths does not need to be configured as they are estimated by the mechanisms. We show comprehensive experiments with FBLB and present performance results. The results include both download and streaming traffic as they exhibit different flow dynamics.

Furthermore, we give recommendations for FBLB configuration parameters.

4.1.1 Performance Comparison of MPTCP and PBLB

In this section, we investigate the performance of simple variants of PBLB and MPTCT. We first describe the testbed and our implementation. Then we present the evaluation scenario and analyse the results.

4.1.1.1 Testbed

We set up four virtual machines (VMs) with a dedicated core on a server: a client VM, a server VM and two link-modelling VMs.

We used Linux bridges so that client and server VM communicated with each other via both link modelling VMs – that is, two paths exist between the client VM and the server VM. We utilised the Linux tc tool [115] to limit the rate of the outgoing interfaces of the link modelling VMs with a token bucket filter with rates c_0 and c_1 of links l_0 and l_1, a bucket size of 3 maximum packet sizes (see recommendation in [2]) and a latency of 10 ms. The latency determines the buffer size. In addition, we utilised the netem module of tc to model delay d_0 and d_1 on these links. The link modelling was done in accordance with the principles described in this work.

In the different experiments, only the software configurations on the client and server VMs were modified. To exchange traffic, we utilised a client program on the client VM and a server program on the server VM. For download experiments, wget version 1.20.3-2 [116] was used as the client program and Busybox httpd 1.30.1-4 [117] as the server program. The design of the streaming experiments will be elaborated in detail when they are introduced in the performance section.

4.1.1.2 Implementation

To experiment with MPTCP, we utilised Linux Kernel 4.19 (v0.95) on the client and server VM as this is the latest version supporting MPTCP (status 19.11.2019).

To experiment with PBLB and later with FBLB, I utilised Linux Kernel 5.1 on the client and server VM to work with an up-to-date version of TCP Cu-

bic. As there is no recommended PBLB mechanism described in the literature, I presented simple methods and Java-based programs for a load balancer (LB) and a recombination function (RF). We further describe their integration into the testbed.

Implementation of the PBLB Load Balancer The PBLB is based upon a simple token bucket. The bucket has a certain capacity and fill rate. Whenever a packet is sent over a link, the amount of tokens in the bucket for that link is reduced by the size of the packet. If there are not enough tokens in the bucket, a packet cannot be sent over that link. The fill rate adds tokens back to the bucket depending on the link speed. The LB receives traffic from a socket and meters it with a token bucket with rate r_{TB} and bucket size s_{TB}. We configure r_{TB} with $0.99 \cdot c_0$ and s_{TB} with $0.97 \cdot b_0$ where c_0 and b_0 are the bandwidth and latency of a link l_0. In-profile traffic is sent to link l_0, and out-of-profile traffic, when there are not enough tokens in the bucket for l_0, is sent to link l_1. Before transmission, packets are equipped with a 4-byte LB header containing a sequence number (SN), an 8-byte UDP header and a 20-byte IP header with an appropriate IP address of the RF (BNG) as the destination. In addition, logging is performed for performance evaluation and debugging purposes.

Implementation of the PBLB Recombination Function The RF receives traffic from the LB. It removes the additional headers, evaluates the SN and immediately forwards in-order packets to their destination. Out-of-order packets are timestamped and buffered. After a timeout t_{out}, they are forwarded with all consecutive in-order packets in the buffer. The timeout is link specific and should be set to the latency of the other link plus the difference of the delay between the other link and this link; 10-ms safety margin are added. We chose this lightweight approach for the timeout for efficiency of implementation. If a new packet arrives, the oldest timeout of the buffered packets is checked, and if it has expired, the packet is forwarded. In theory, this may lead to slightly longer waiting times in the buffer, but they were negligible in our experiments.

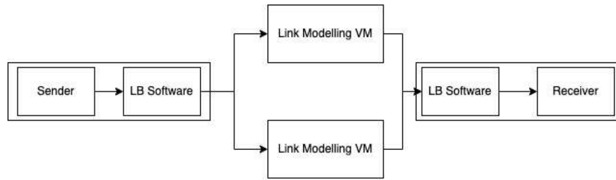


Figure 4.1: *VM setup of the hybrid access experiments.*

Integration of LB/RF in the Testbed In our testbed, the LB/RF programs were also activated on the client and server VMs. That is, they were collocated with the client and server programs, and I did not model real HG/BNG as they do not influence the performance of the load balancing algorithms. Those components are used by providers to integrate customers into their network; however, they only add a slight latency that is part of the overall link latency for our experiments. The client/server program and LB/RF utilise tun interfaces to exchange traffic on layer 3 within the same VM. That means that the client/server program sends its traffic to a tun interface which is read from the LB. The LB then load-balances the traffic over two paths towards the RF. The RF reads data from these paths and writes them to a tun interface which is read from the TCP server. The setup is visualised in Figure 4.1

4.1.1.3 Performance Comparison

We first explain the experiments and then we discuss results for MPTCP and PBLB.

Experiment Design In the experiment, we simultaneously downloaded $n \in \{1, 2, 4, 8, 16, 32\}$ files of size 10 MB and measured the required time t_{load}^{down} until the completion of the last download. We determined the relative goodput by $\frac{n \cdot 10\text{MB}}{t_{load}^{down} \cdot (c_0 + c_1)}$, where c_0 and c_1 were the bandwidths of both links. We ran each experiment 10 times and reported average values for the relative goodput.

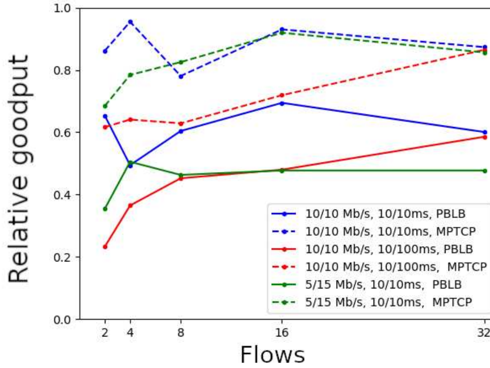


Figure 4.2: *Relative goodput for MPTCP and TCP over PBLB.*

To determine the download time with MPTCP, the time from starting wget until completion was measured. In the case of TCP over PBLB or FBLB, we evaluated the logs of the LB as this is simpler and leads to more accurate results.

Performance of MPTCP We performed the above experiment series for MPTCP and TCP Cubic over PBLB over three different HA scenarios: (1) l_0 and l_1 were both configured with 10 Mb/s and 10 ms delay, (2) like (1) but l_0 and l_1 were configured with 5 Mb/s and 15 Mb/s, (3) like (1) but the delay of l_1 was set to 100 ms. The relative goodput for these experiments is compiled in Figure 4.2.

With MPTCP over equal links (1), a relative goodput of around 90% was achieved for flow counts of 1, 2, 4, 8 and 32. Over asymmetric links with equal delay but bandwidth 5/15 Mb/s (2), a clearly lower relative goodput of 60% was achieved when the number of flows was small, but the relative goodput increased with the number of flows. Over asymmetric links with equal bandwidth but delay 10/100 ms (3), the relative goodput was again rather large for 4 and more flows.

Performance of PBLB With TCP Cubic over PBLB, the relative goodput was consistently lower than with MPTCP in all investigated scenarios. The difference in relative goodput is significant, while the overhead through additional headers for LB amounts to only about 2%.

We suppose that the performance degradation results from jitter introduced by the RF when packets are buffered to restore their original order. This creates micro bursts that lead to early packet loss. Moreover, with PBLB, the capacity and buffer of link l_0 cannot be fully leveraged by design. This is only a brief analysis and more investigation is needed.

We have implemented modifications of the presented PBLB variant that reduce jitter and increase relative goodput. However, their benefits are not sufficient. Thus, alternatives to PBLB for HA are desirable, and so we consider FBLB an option that is worth further investigation.

4.1.2 Flow-Based Load Balancing

In this section we suggest a framework for FBLB that we limit to two available links for the sake of simplicity. For flow-based load balancing, no permanent recombination is required, only the distribution between links. Therefore, it is possible to utilise more than two links without introducing additional overhead. The LB measures the available bandwidth of both links and the current rates of all active flows and utilises this information to assign new flows to one of both links. It detects overloaded links and reassigns flows from the more overloaded link to the other. During the process of rebalancing a flow from one link to another, a flow can possibly be assigned to a link with a lower latency as its origin link. In this case, packet reordering can occur as long as there are still in-flight packets on the old links. During this short period of time, an additional load balancing header carrying the packet sequence number is added to each packet. For packets with a sequence header, the same RF that is used for PBLB is utilised to remove the header and forward the packets in order.

We first describe measurement methods for flow- and link-related metrics. We

then propose and test algorithms for flow assignment to a link and finally suggest a method for flow reassignment, including protocol support.

4.1.2.1 Measurement Algorithms

The LB keeps statistics for each flow and link. They are collected on the basis of measurement intervals with an equal length of T_{MI}. The statistics are updated upon packet arrival or at the ends of measurement intervals.

Flow-Specific Metrics For each flow f, an entry is stored in a hash map. The entry contains the number of bytes b_f received within the current measurement interval. At the end of a measurement interval, the current flow rate, $r_f = \frac{b_f}{T_{MI}}$, is computed, and b_f is reset to zero. In addition, the number of consecutive intervals without any packet arrival is recorded by the variable z_f. After z_{max} such intervals, a flow is considered terminated and removed from the statistics.

Upon the arrival of an IP packet with size s, the hash for the corresponding flow is computed. The hash takes the source and destination IP address, the source and destination port numbers and the protocol number into account. If an entry exists for that flow, b_f is updated. Otherwise, the packet belongs to a new flow. Then, a new entry is established in the hash map and initiated – that is, $b_f = 0$, $r_f = 0$ and $z_f = 0$ – and the flow is assigned to a link, which is recorded by l_f. We suggest several algorithms for that task in Section 4.1.2.2. Finally, the packet is forwarded over the interface indicated by l_f.

Bandwidth Estimation I assume that the bandwidths of the links are fairly stable but unknown and describe a simple measurement mechanism to derive their capacity. The number of bytes b_l forwarded over a specific link l within the current measurement interval is stored and updated with every packet transmitted over that link. At the end of a measurement interval, the link load is computed by $r = \frac{b_l}{T_{MI}}$, and b_l is reset to zero. In addition, the two most recent link loads are recorded by r_l^0 and r_l^{-1}. An estimate of the available bandwidth of a link is

initially set to a low value of $c_l^{est} = c_{init}$, and the last two link loads are initialised with zero. It is updated as follows. When r_l^0 and r_l^{-1} are updated at the end of a measurement interval, the current rate is checked for stability – that is, whether $\frac{|r_l^0 - r_l^{-1}|}{c_l^{est}} < T_{stability}^{rate}$ is fulfilled. As a result, the rate stability threshold $T_{stability}^{rate}$ is a real number close to zero, such as $T_{stability}^{rate} = 0.01$. If the current rate is stable and larger than the last estimate of the available bandwidth – that is, if $\frac{r_l^0 + r_l^{-1}}{2 \cdot c_l^{est}} > T_{exceeded}^{bandwidth}$ holds – the estimate of the available bandwidth is updated by $c_l^{est} = \frac{r_l^0 + r_l^{-1}}{2}$. As a result, the bandwidth-exceeded threshold $T_{exceeded}^{bandwidth}$ is a real number close to 1; for example, $T_{exceeded}^{bandwidth} = 1.00$. The rationale behind this approach is that spikes in link loads should not lead to overestimated link bandwidths. Such spikes may occur due to bursts received at rates higher than the actual link bandwidths. This works sufficiently well for links with stable bandwidths. For links with volatile bandwidths, this can still be utilised to gain an estimate about the average bandwidth. Short changes in bandwidth need to be dealt with on Layer 4 by, for example, TCP.

The measurement system can be compromised by traffic being sent at a rate higher than the actual bandwidth. There are easy countermeasures, but they are not part of this work. Detecting attacks in a series of averages is a problem solved in statistics, not networking.

Number of Flows Furthermore, the number of a link's active flows, a_l, is incremented when a flow is assigned to a link, and it is decremented when an existing flow is reassigned to the other link or when it is removed from the statistics.

4.1.2.2 Flow Assignment Strategies

We suggest three strategies for assigning a flow to a link when the first packet of a flow arrives at the LB. After flow assignment, the assignment instant t_f^{assign} is set to the current time t_{now}.

Equal Flow Density Flow Assignment The objective of this strategy is to balance flows over the links such that the number of flows on the links is proportional to their capacities. If a new flow f arrives and $\frac{a_{l_0}}{c^{est}_{l_0}} \leq \frac{a_{l_1}}{c^{est}_{l_1}}$ holds, the new flow is assigned to link l_0 so that the link of the new flow is initialised with $l_f = l_0$ and the number of active flows a_{l_0} on l_0 is incremented by 1. Otherwise, the new flow is assigned to l_1 with analogous operations.

Equal Utilisation Flow Assignment The objective of this strategy is to balance flows over the links such that the sum of the flow rates of the links is proportional to their capacities. If a new flow f arrives and $\frac{r^0_{l_0}}{c^{est}_{l_0}} \leq \frac{r^0_{l_1}}{c^{est}_{l_1}}$ holds, the new flow is assigned to link l_0 so that the link of the new flow is initialised with $l_f = l_0$ and the number of active flows a_{l_0} on l_0 is incremented by 1. Otherwise, the new flow is assigned to link l_1 with analogous operations.

Token Bucket Control Flow Assignment Token bucket control (TBC) adopts the principle of 'threshold metering' [118] to determine whether link l_0 is overloaded. A token bucket's rate r_{TB} is configured with $c^{est}_{l_0}$. I denote the maximum value of the token bucket by F_{max} and its minimum value by F_{min}. The last update instant is recorded by the variable t_{last}.

When a packet arrives, the fill state F of the token bucket is incremented by $F = min(F_{max}, F + r_{TB} \cdot (t_{now} - t_{last}))$, and the variable t_{last} is updated by the current time t_{now}. If the packet belongs to a new flow and $F \geq F_{th}$ holds, link l_0 is not overloaded. In that case, the new flow is assigned to l_0 – that is, $l_f = l_0$ – and the number of active flows a_{l_0} on l_0 is incremented by 1. Otherwise, the new flow is assigned to link l_1 with analogous operations. Finally, any arrived packet is metered if it belongs to l_0. That is, the fill state F is decremented by the packet size s; that is, $F = F - s$.

4.1.2.3 Flow Reassignment

At the end of a measurement interval, the LB checks for overload on the links. I consider a link overloaded if

$$\frac{r_l^0}{c_l^{est}} \geq T_{load}^{over}. \tag{4.1}$$

holds based on the newly computed estimate of the available link bandwidth c_l^{est}. The impact of different T_{load}^{over} will be shown by experiment. In case of overload, a flow may be moved to another link. If both links are overloaded and $\frac{r_{l_0}^0}{c_{l_0}^{est}} > \frac{r_{l_1}^0}{c_{l_1}^{est}}$ holds, the load on l_0 is larger than on l_1, so one flow should be moved from l_0 to l_1. Otherwise, one flow should be moved from l_1 to l_0. In the following section, we explain methods for determining appropriate flows to be moved from one link to the other and protocol support to avoid packet reordering when a flow is reassigned to another link.

Determining a Flow for Reassignment from l_0 to l_1 We determine a flow on link l_0 to be reassigned to link l_1. Only flows with $t_f^{assign} \leq t_{now} - T_{guard}$ are eligible. T_{guard} is a value that needs to be determined by experiment for each flow type, as it is highly dependent on how the application reacts to jitter and changes in bandwidth and/or latency. For our downloads and synthetic streaming tests, we determined a value of T_{guard} of 2 s to be high enough to allow TCP to adapt to a new link.

If there is no eligible flow, no flow is reassigned. Otherwise, the eligible flows are ordered according to descending rates r_f. Then, the sum of their rates is incrementally computed based on this flow order and the first flow, for which the rate sum exceeds c_{l_0}, is selected to be moved to l_1.

Determining a Flow for Reassignment from l_1 to l_0 We determine a flow on link l_1 to be reassigned to link l_0. Only flows with $t_f^{assign} \leq t_{now} - T_{guard}$ are eligible. If there is no eligible flow, no flow is reassigned. Otherwise, the flow with the smallest rate is reassigned from l_1 to l_0.

Protocol Support to Avoid Packet Reordering When a flow is reassigned to another link, packets may be reordered if they are more quickly delivered on the new link than on the old link. As packet reordering may be detrimental to some applications, I propose protocol support to avoid packet reordering.

When the LB reassigns a flow to another link, it adds LB headers with SNs to packets of that flow for the next T_{out} time. Those headers and the value for T_{out} are the same as those described in Section 4.1.1.2. The RF resequences packets that arrive with an LB header, just as in Section 4.1.1.2. When a flow is reassigned, the LB sends an empty packet with an LB header to the RF over the old link of the reassigned flow. Thereby, the RF learns the last SN, and packets of the reassigned flow are numbered with higher SNs, which serves for synchronisation. Generally, a packet is equipped with an LB header if $t_{now} - t_f^{assign} \leq T_{out}$ holds for its flow. As LB headers are added only for T_{out} times after flow reassignment, their protocol overhead is marginal.

The LB header may reduce the MTU size on the links between LB and RF during reordering, which possibly causes MTU issues. Therefore, the MTU should be set to a low enough value to avoid such problems.

4.2 Performance Evaluation of FBLB

In this section I evaluate the performance of FBLB and compare it to PBLB and MPTCP. I utilised the same testbed as described in Section 4.1.1.1 and integrated LB/RF for PBLB/FBLB as outlined in Section 4.1.1.2.

We first illustrated the load-balancing effect of FBLB with appropriate parameters for download traffic and for streaming traffic in low- and high-load conditions.

Then, we conducted experiments to determine reasonable system parameters for FBLB.

We report the impact of the suggested flow assignment methods for download and streaming flows.

Finally, we compared the download times and achieved streaming rates for FBLB, PBLB and MPTCP under different loads.

If not mentioned otherwise, we used the following parameters in the experiments: measurement interval duration $T_{MI} = 100$ ms, overload threshold $T_{load}^{over} = 1.00$, recombination timeout $T_{out} = 500$ ms, guard time for flow reassignment $T_{guard} = 2$ s and TBC's maximum bucket size $F_{max} = 0.97 * c_{l_0}$. Furthermore, we performed experiments with two equal links with a capacity of 10 Mb/s and a one-way delay (OWD) of 10 ms.

4.2.0.1 Illustration of FBLB

To provide a qualitative understanding of FBLB, we first illustrate its behaviour for download and streaming traffic.

FBLB Behaviour for Download Traffic We subsequently started 8 downloads of files with 10 MB with a 5-s inter-start time. Figure 4.3 shows their measured rates over time. The rates were obtained at the RF and include all header overheads. Each colour belongs to one flow. The curves above the *x*-axis correspond to flows on link l_0, and the curves below correspond to flows on link l_1. The uppermost and lowermost curves indicate the overall rate on each link. We observed that flows subsequently started on different links. The fifth flow was reassigned after a short time. At that time, the overall rate temporarily exceeded the link bandwidth, which was the trigger for reassignment. At the beginning of the experiment, the flow rates were large, and they decreased with the increasing number of flows. When downloads completed, the flow rates increased again.

FBLB Behaviour for Streaming Traffic We first describe our traffic generator for streaming traffic. Then I illustrate load balancing with streaming flows under high-load and overload conditions.

Generation of Streaming Flows The tool iperf3 [119] is used in many performance studies to generate streaming flows. However, it is also known to

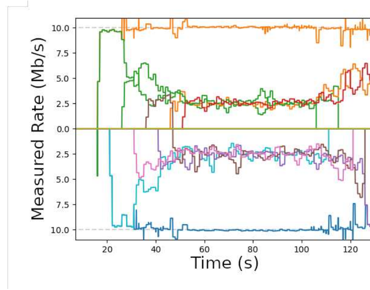
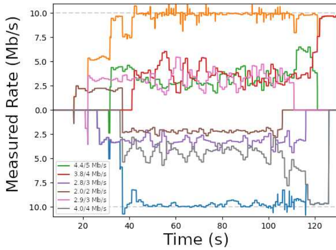
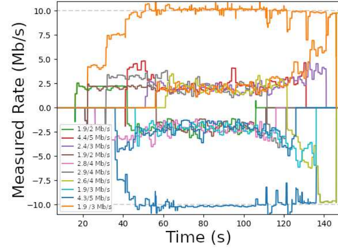


Figure 4.3: *8 download flows are load-balanced by FBLB to two links; with a capacity of 10 Mb/s and a OWD of 10 ms; flows above the x-axis correspond to link l_0, flows below the x-axis to l_1.*



(a) High-load condition: 5 streaming flows with 2, 3, 4, 5 and 6 Mb/s.



(b) Overload condition: 10 streaming flows which are twice the flows as in high-load condition.

Figure 4.4: *Streaming flows are load-balanced by FBLB to two links; with a capacity of 10 Mb/s and a OWD of 10 ms; flows above the x-axiscorrespond to link l_0, flows below the x-axis to l_1.*

create overly bursty traffic [120]. Therefore, we programmed an own generator

for streaming flows in Java. A streaming flow is transmitted via TCP with a desired rate r_{app} on the application layer. In case of congestion, a lower data rate is transmitted. To that end, we fill the send buffer of a TCP connection every 20 ms with $r_{app} \cdot 0.02$ s bytes if the fill state of the send buffer is below 50%. We used the Linux default value of 87380 bytes for the send buffer size. The duration of the streaming flows was limited by the time value instead of the transmitted data volume.

High-Load Condition We first considered FBLB with 6 streaming flows having application rates of 2, 3, 3, 4, 4 and 5 Mb/s, which is 21 Mb/s in sum. That means that with protocol overhead, there is only slight congestion if the bandwidth of both links can be fully utilised and we consider this a high-load condition. Figure 4.4(a) illustrates the measured rates of the 5 streaming flows over time. The measurement methodology was the same as in the previous experiment. The streaming flows were subsequently started; they had different application rates r_{app} and almost achieved them as measured rates. When some flows were complete, the remaining flows fully utilised the available bandwidth since their TCP send buffer had been filled during the congestion phase before.

Overload Condition We now nearly double the amount of flows so that their overall application rate is 35 Mb/s, which is nearly twice the overall bandwidth of both links combined. This is a clear overload condition. Figure 4.4(b) illustrates the rates of the 10 streaming flows over time. They are subsequently started, and as soon as there is overload on the link, they contend for bandwidth and their rates are throttled to their fair share. When flows complete at the end of the experiment, the remaining flows utilise the remaining bandwidth. Now we see reassignment events during the start phase and the completion phase of the flows.

4.2.0.2 Impact of the Measurement Interval T_{MI}

The measurement methods in Section 4.1.2.1 utilise disjoint intervals. We investigate the impact of the interval duration on the measured rate. To that end, the

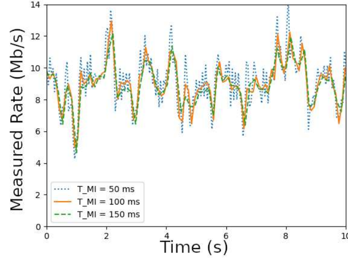


Figure 4.5: *Impact of measurement interval duration T_{MI} on the measured rate of a single flow; 2 download flows are carried over a single link. with 10 Mb/s and a OWD of 100 ms.*

client downloads traffic from the server using 2 download flows over a single link with a capacity of 10 Mb/s and a OWD of 100 ms. Figure 4.5 illustrates the rate of one of these flows with different interval durations T_{MI}. A short measurement interval of 50 ms causes highly variable flow rates. A long measurement interval of 150 ms leads to more stable flow rates but also to more delayed results as the rates are available only at the end of the intervals. A measurement interval of 100 ms seems long enough to produce sufficiently stable rates while they are delivered with moderate delay. The investigated scenario can be considered a challenge because measured rates for an OWD shorter than 100 ms or with more flows are more stable.

Thus, in the following, we configure the load balancing algorithm with a duration of the measurement interval of $T_{MI} = 100$ ms.

4.2.0.3 Impact of the Overload Threshold T_{load}^{over}

Flow reassignment at the LB is triggered by overload detection according to Equation (4.1). It requires the parameter T_{load}^{over}, for which we derive a recommendation in the following experiment. We download two files over FBLB and

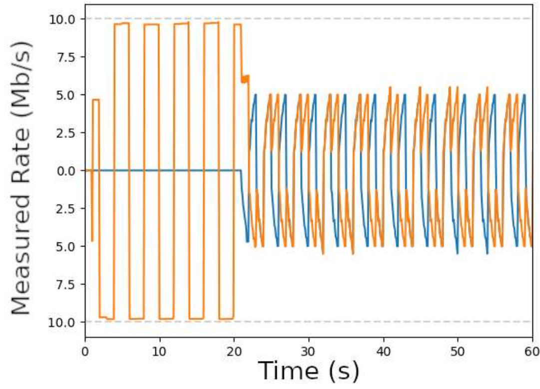


Figure 4.6: *Two greedy flows with an overload threshold of 0.99.*

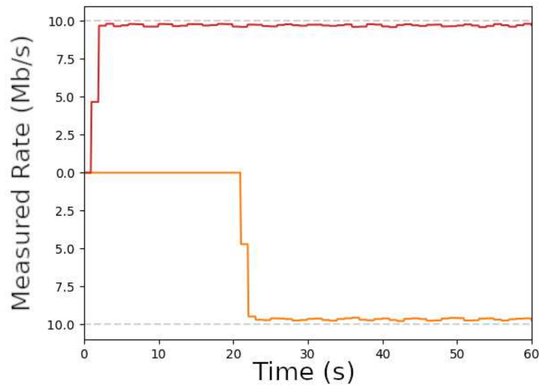


Figure 4.7: *Two greedy flows with an overload threshold of 1.00.*

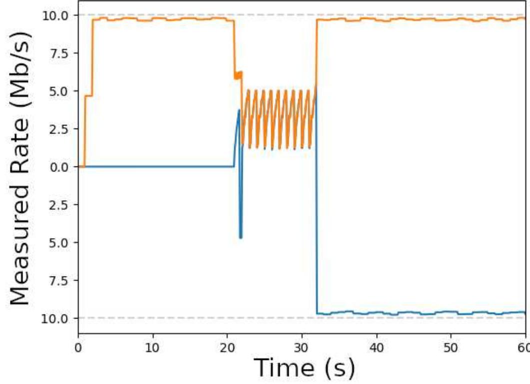


Figure 4.8: *Two greedy flows with an overload threshold of 1.01.*

two links and start the second download 3 s after the start of the first download. Both flows are intentionally assigned to the first link after the start. I test $T_{load}^{over} \in \{0.99, 1.0, 1.01\}$, for which the results are compiled in Figures 4.6–4.8. Figure 4.6 shows that $T_{load}^{over} = 0.99$ leads to oscillations. This overload threshold is easily exceeded by traffic so that flows can be reassigned again as soon as their guard time T_{guard} is over after (re-)assignment. Both the first and the second flow are continuously reassigned, which does not efficiently utilise the capacity of both links.

Figure 4.7 shows that $T_{load}^{over} = 1.0$ avoids oscillations and assigns one flow per link shortly after the second flow starts.

Finally, Figure 4.8 shows that $T_{load}^{over} = 1.01$ avoids oscillations but takes a long time until one flow is reassigned to the other link as overall link rates are mostly too slow to exceed the overload threshold.

The load balancing algorithm works well with $T_{load}^{over} = 1.00$, so we recommend that value for operation.

Impact of Flow Assignment Strategies We investigated the impact of the different flow assignment strategies presented in Section 4.1.2.2 and briefly summarise the results. We tested them while deactivating the flow reassignment.

In one experiment, we started 10 downloads simultaneously. The expectation was that equal utilisation (EU) and TBC could not distribute flows to other links because they require appropriate measurement results, which are missing in case of simultaneously starting downloads. However, we were unable to produce this phenomenon as the simultaneous start turned out to be a challenge; EU and TBC achieved good traffic distribution even though a bit worse than equal flow density.

In another experiment, we subsequently started streaming flows. They had different rates with an overall rate close to the overall bandwidth of both flows. Here, EU achieved the best traffic distribution in that the flows reached a maximum overall streaming rate. TBC revealed the worst result in this experiment but maximised the fraction of traffic on the first link.

As soon as we activated flow reassignment, I obtained similar results for all three flow assignment strategies. In the following, we apply EU.

4.2.0.4 Performance Comparison for Various Load

In the following, we compare the performance of FBLB with flow reassignment, PBLB and MPTCP for download and streaming traffic over two links. We use transmission over a single link with single and double bandwidth (10/20 Mb/s) as benchmarks and consider different load scenarios.

Evaluation with Download Traffic We first explain the traffic model and then discuss performance results.

Traffic Model Download requests for files with a size of $B = 20$ MB start with interarrival times A, which are determined by a exponential distribution.

We control the relative system load ρ by setting the mean of the interarrival

time to

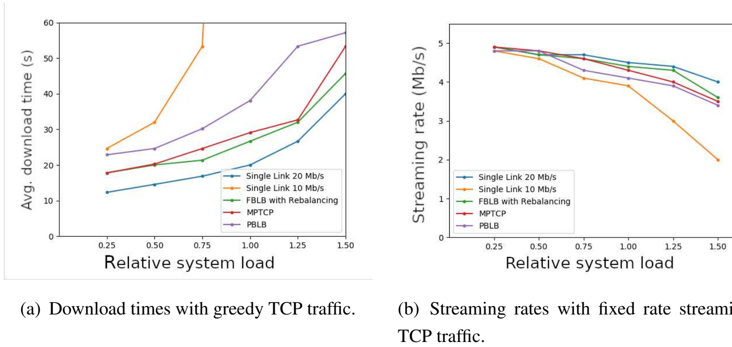
$$E[A] = \frac{B}{\rho \cdot c}. \quad (4.2)$$

The capacity c is the bandwidth of a single link with a bandwidth of 10 Mb/s. The system load ρ as defined in this context relates to application layer traffic and ignores overhead on lower layers. Therefore, the real bandwidth utilisation is higher than ρ – that is, a system with a load close to 100% is already overloaded.

Experiment Design We considered relative system loads $0.25 \leq \rho \leq 1.5$ based on a link with the combined capacity of both bundled links. For each system load we generated 10 traces over 10 minutes of download requests according to the presented traffic model. Then we conducted experiments for the benchmark and each considered load balancing method. We took the average download time as the performance metric.

Performance Comparison Figure 4.9(a) shows the average download time depending on the system load. It increases with system load and is shortest when traffic is downloaded using single-path TCP over a single link with a double bandwidth. It is drastically larger for a single link with a single bandwidth, and the system seems already overloaded at a relative load of 75%. With MPTCP over two links, the download time is larger than over a single link with double capacity. However, significantly more load can be supported than with single-path TCP over a single link. Thus, MPTCP can well combine the bandwidth of both links. PBLB can support similar load levels but causes clearly larger download times. Reasons for this inefficiency are mentioned in Section 4.1.1.3 and were the motivation to develop FBLB. Download times for FBLB are even slightly shorter than with MPTCP for most load levels.

Evaluation with Streaming Traffic We now investigate traffic streaming in a similar setting. The main difference is the traffic model.



(a) Download times with greedy TCP traffic.

(b) Streaming rates with fixed rate streaming TCP traffic.

Figure 4.9: *Impact of system load when traffic is load-balanced over two links with different methods compared to a single link with the combined bandwidth.*

Traffic Model We generate streaming flows with application rates of 2 Mb/s with 25% probability, 5 Mb/s with 50% probability, and 8 Mb/s with 25% probability. Thus the average application rate is 5 Mb/s. Every flow takes 32 s so that flows produce a traffic volume of up to 8 MB, 20 MB and 32 MB on the application layer with an average of 20 MB[1]. As in Section 4.2.0.4, I choose exponentially distributed interarrival times and calculate their average $E[A]$ using Equation (4.2) with $E[B] = 20$ MB, depending on the desired relative system load ρ.

Performance Comparison Figure 4.9(b) shows the achieved streaming rates depending on system load ρ. They increase with system load. They are smallest for single-path TCP over a single link with single capacity, and they are largest for single-path TCP over a single link with double capacity. MPTCP transmitting over the two links well approximates the streaming rates achieved

[1]If the full application rate r_{app} cannot be achieved, less traffic is streamed over time.

for the single link with double capacity. PBLB achieves lower streaming rates than MPTCP. With FBLB the streaming rates are mostly slightly larger than with MPTCP.

5 Software Defined Networking

This chapter contains an overview of P4-SFC which was also published at the High Performance Computing conference in 2020

5.1 Architecture of P4-SFC

NFV and SFC are a major component in future cost reduction for infrastructure providers. It allows them to efficiently use off-the-shelf hardware to emulate network functions like firewalls, proxies, etc. It is possible to implement a part of those functions in P4, however the knowledge about P4 is nearly non present with datacenter operators. There is currently no framework to allow normal developers and administrators to leverage the possibilities of NFV/SFC. With this work we want to change that. In this chapter we explain the implementation of a P4-based SFC system and how VNFs are efficiently integrated on hosts so that they remain SFC-unaware and transparent for routing. The system consists of three parts. The SFC aware Ingress node that determines the path through the network, the MPLS overlay network that is used for forwarding, and the servers that are used to emulate the network functions.

5.1.1 Implementation of the SFC Ingress Node

We briefly describe the requirements of the SFC ingress node and explain a P4 pipeline for implementation of this functionality in P4.

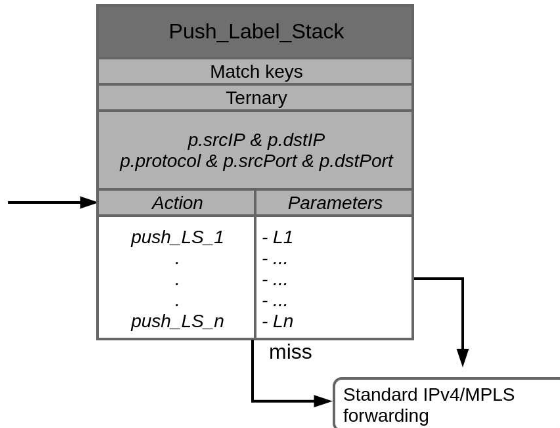


Figure 5.1: *Match-and-action table "push_Label_Stack" to push label stacks of different size.*

5.1.1.1 Requirements

In P4-SFC, the SFC ingress node classifies traffic and adds appropriate MPLS label stacks to packets that require processing by a specific SFC. The classifier identifies flows for a specific SFC. We utilize flow descriptors consisting of source and destination IP addresses, port numbers, and IP protocol number for that purpose. Wildcards are supported. That label stack encodes both the forwarding in the network and the identification of the VNFs. Therefore, the label stack can be large. To keep things simple, we support support up to $n = 10$ labels in our small testbed (see Section 4.1.1.1). However when using jumbo frames, large numbers of labels are possible.

5.1.1.2 P4 Pipeline

We describe the supported header stacks, the ingress and egress control flow, and the pushLabelStack control block in more detail.

Supported Header Stacks Incoming packets are parsed so that their header values can be accessed within the P4 pipeline. To that end, we define up to n MPLS labels, an IP header, and a TCP/UDP header. It is important to define different pattern schemes for different amounts of Labels as the parsing in line speed does not allow for a dynamic recognition of the next header field in the packet.

Ingress and Egress Control Flow The ingress control flow consists of a Push_Label_Stack control block and an IP_MPLS_Forward control block. The push_Label_Stack control block adds an appropriate label stack to the packet, i.e., it serves as classifier. The IP_MPLS_Forward control block performs simple IP/MPLS forwarding. The egress flow just sends the packet and does not implement any special control blocks.

Implementation of the Push_Label_Stack Control Block The implementation challenge is that an arbitrary number of up to n labels need to be pushed. Header sizes are fixed. An intuitive approach is pushing a single label per pipeline execution and recirculating the packet for another pipeline execution until the desired label stack is fully pushed. The drawback of this approach is that pushing n labels requires n-fold packet processing capacity, which reduces the throughput of the SFC ingress node.

Our solution uses the match-and-action table (MAT) push_Label_Stack whose structure is given in Figure 5.1. The MAT utilizes the fields of the source and destination addresses and port numbers as well as the IP protocol number as match keys. A ternary match is used so that wildcards are supported. We provide actions push_LS_i to push a stack of i labels onto the packet. This action has i parameters but the table has n label entries (L1, ..., Ln). In case of a match, the corresponding action is executed with the appropriate number of arguments. Afterwards, the IP_MPLS_Forward control block is carried out. For the implementation of the IP_MPLS_Forward control block we reuse available demo code.

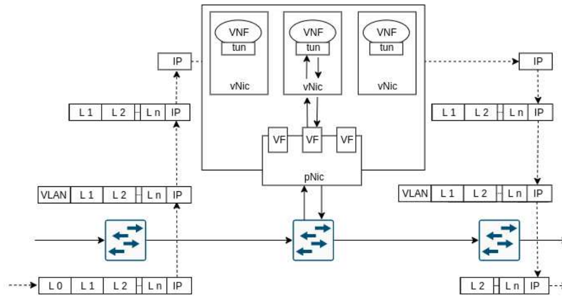


Figure 5.2: *P4-SFC utilizes label stacks in packets for segment routing in the network, but passes only IP packets to VNFs. Therefore, VNF remain unaware of the SFC.*

5.1.2 Transparent and Efficient VNF Integration on Hosts

We now specify how packets are forwarded from a switch to a VNF on a host and back. This is challenging since the VNF should remain unaware of the label stack, and the packet forwarding from the host's network interface card (NIC) to the VM hosting the VNF should be efficient. The following steps are illustrated by Figure 5.2

We assume that up to N VNFs are supported by a host, either within a container or a separate VM. Each potential VNF constitutes a logical network segment while the corresponding physical network segment is the switch over which the VNF is reachable. The forwarding table of the switch is configured such that an incoming packet with a topmost label pointing at a specific VNF is equipped with a VLAN tag pointing at the VNF and forwarded to the respective host.

The NIC of the host is statically configured to map packets with VLAN tags to virtual PCI devices that serve as virtual NICs (vNICs) for VMs or containers. These features are enabled by virtual machine device queue (VMDq) and single root I/O virtualization (SR-IOV). These technologies are supported by most

contemporary NICs and CPUs. With VMDq, a NIC can have multiple internal queues and with SR-IOV, a so-called Physical Function (PF) can be virtualized into Virtual Functions (VFs). A VF can be passed-through as PCI device to a VM or container. We utilize SR-IOV to pass through a queue of the NIC as a VF to a VM/container in order to serve as a vNIC. The NIC used in our prototype provides up to 128 VFs so that up to $N = 128$ VNFs can be supported on a host. More powerful NICs providing even more VFs also exist.

Within a VM/container, the forwarding table of the MPLS Router Module in the Linux kernel is utilized to deliver the IP packet to the VNF without the label stack, to store the label stack, to push the label stack again when the packet is returned from the VNF, and to send the packet to the appropriate egress interface. Then, the packet is returned from the host to the switch in the corresponding VLAN. The switch removes the VLAN tag and the label for the next segment.

5.1.3 P4-SFC Orchestrator

The P4-SFC orchestrator is written in Python and leverages the libvirt and LXD framework for VM/container management. It interacts with administrators for management purposes and with customers for the specification of SFCs. It places VNFs on hosts and computes paths for SFCs, it launches and terminates SFCs, it adds new hosts and migrates VNFs among hosts.

5.1.3.1 Administrator/Customer Interaction and SFC Specification

The orchestrator offers a CLI interface for maintenance, e.g., for adding a new host to the system or moving VNFs.

Customers provide a configuration file in json format with a description of their SFCs. The specification of an SFC includes a flow descriptor, a list of VNFs, their executable binaries, their resource requirements (CPU, RAM, I/O), and information whether they are to be deployed as VMs or containers. Customers may request permanent storage for a VNF, e.g., for logging purposes, so that it has permission to write to shared network storage. The VNF binaries provided by cus-

tomers are also saved to shared network storage. VNF applications are required to receive and send packets via /dev/net/tun, but they can remain unaware of SFCs.

5.1.3.2 VNF Placement and Path Calculation

The orchestrator determines hosts to run VNFs such that resource requirements communicated by the customers are met. Storage is not part of these requirements since shared network storage is used. If resources are not sufficient, VNFs may be migrated or new hosts may be added. While there is an extensive body of literature on VNF placement, our prototype uses only simple algorithms for this task.

The orchestrator knows the network topology. Either the network topology is static like in our prototype or it can be dynamically discovered with protocols like LLDP [121]. Based on this information, the orchestrator computes paths from the SFC ingress node to desired destinations including the VNFs specified by SFCs. The path calculation is performed whenever a forwarding entry for the SFC ingress node needs to be modified.

5.1.3.3 Launch and Termination of SFCs

The orchestrator holds a disk image as template for VMs/containers supporting VNFs. P4-SFC requires an appropriate configuration of the forwarding table of the MPLS Router Module which is initially applied to the template. The template is copied to every host so that VMs/containers can be cloned from it. Resources required by a VNF are provided by the customer's SFC description and are enforced by the orchestrator using appropriate configuration files for the VM/container. A libvirt xml definition specifies the hardware resources assigned to a VM. Similarly, an LXD configuration file uses cgroup statements to limit the kernel space resources available to the container.

If an SFC is to be launched, the orchestrator determines for each VNF a host with sufficient resources and finds a free VF on the NIC of that host. This VF

determines the label for the VNF. The orchestrator defines a VM/container with suitable parameters, i.e., the VM/container template, sufficient resources, the VF, and a pointer to the VNF binary. It then starts the VM/container and the appropriate VNF binary from the shared network storage. Finally, the SFC ingress node is configured. To that end, a path is computed for the SFC and an entry is added to the MAT push_Label_Stack (see Figure 5.1) containing the flow descriptor and the label stack for the SFC. The flow descriptor is needed for packet classification.

If an SFC is to be stopped, the VMs/containers with its VNFs are terminated and the corresponding entry is removed from the MAT push_Label_Stack.

5.1.3.4 Adding a New Host

To add a new host to P4-SFC, the orchestrator needs ssh access and permissions for VM/container management on the new host. It initially scans for available resources on the new host and adds them to its pool of available capacities. It then copies the VM/container templates to the host and configures the virtualization frameworks.

To make the new host and its potential VNFs reachable in the network, the forwarding table of the switch to which the host is attached is equipped with forwarding entries for the labels of all potential VNFs on the new host. If a host is removed, the corresponding labels are removed from the switch.

5.1.3.5 VNF Migration

VNFs may need to be migrated to another host, e.g., for maintenance purposes. The orchestrator supports this process by first cloning and starting the VNF on the new host, changing the respective entry of its SFC in the MAT push_Label_Stack, and terminating the VNF on the old host.

6 Conclusion

The core component of this work is the VITO framework, a virtualization framework with the goal of replicating physical networks with real hardware as close as possible. VITO can be run on any server with a Linux operating system and can create flexible topologies based on virtualization technologies. In this work I showed how VITO can be used to fully automate all steps of testing networking protocols. VITO can be used to define the topology of the network used for testing, the execution of actions during the test, the collection of results and the repetition and orchestration of tests on available hardware. It is shown how the link modelling process achieves results close to real hardware, even though this introduces additional overhead in the form of link modelling VMs. The foundation of VITO is also used for the semi-virtualized testbeds of the *Internet Praktikum* at the University of Tuebingen. In this scenario, it provides a set of scripts to generate a pre-defined networking topology, consisting of clients, servers and emulated cisco routes with modelled link characteristics to give students an overview of the behaviour of certain software and protocols. This helps to prepare students for challenges and problems which are not expected but occur on real hardware, e.g. bugs in the Linux TCP implementation. This work also presents P4-SFC which extends and modifies the core of VITO to provide a novel framework for enabling NFV and SFC legacy data-centers with minimal costs involved. For this approach, the core orchestration and virtualization part of VITO is reused and enhanced with an MPLS based forwarding data plane. Even if NFV/SFC is not widely used in data-centers yet, P4 NFV/SFC is a foundation early adopters can build on without big investments or the need for experts on the topic.

While hybrid access was an important topic when this work was begun, it is definitely not a technology for the future; rather, it is more of an interim solution for covering up bad ISP infrastructure, especially in rural areas. With the generally good availability of 5G networks and the progress of getting fibre access to the end consumer, there will be few if any uses for this technology, with all its drawbacks, in the future.

Acronyms

SDN	software-defined networking	13
OF	OpenFlow	27
ONF	Open Networking Foundation	31
NFV	network function virtualization	2
CPU	central processing unit	4
VM	virtual machine	3
OS	operating system	3
RAM	random access memory	4
I/O	input/output	12
KVM	kernel virtual machine monitor	6
VMM	Virtual Machine Monitor	6
MMU	memory management unit	8
NIC	network interface card	9
PCI	Peripheral Component Interconnect	10
VT	Intel virtualization technology	10
APICv	Advanced Programmable Interrupt Controller virtualization	11
SLAT	Second Level Address Translation	11
EPT	Extended Page Tables	11
IOMMU	I/O memory management unit	12
DMA	direct memory access	12
BIOS	Basic Input/Output System	12
ATS	Address Translation Service	13
PRI	Page Request Interface	13
ACS	Access Control Services	13

VMD-q	virtual machine device queues	13
QoS	Quality of Service	13
SR-IOV	single root i/o virtualization	14
PCIe	Peripheral Component Interconnect Express	14
PF	physical function	14
VF	virtual function	14
GUI	graphical user interface	15
CLI	command line interface	15
XML	eXtensible Markup Language	15
VW	virtual workspace	49
PW	physical workspace	49
SVM	student VM	52
IVM	infrastructure VM	52
INI	infrastructure network interface	54
LVM	logical volume manager	59
SNMP	Simple Network Management Protocol	28
NETCONF	NETwork CONFiguration protocol	30
MAC	media access control	29
PPP	Point to Point Protocol	24
MPLS	MultiProtocol Label Switching	18
API	application programming interface	29
REST	representational state transfer	29
NBI	northbound interface	29
SBI	southbound interface	30
YANG	yet another next generation	30
RPC	remote procedure call	30
IETF	Internet Engineering Task Force	
LISP	Locator/Identified Separation Protocol	18
DSL	digital subscriber line	22
LTE	Long-Term Evolution	22
P4	Programming Protocol-Independent Packet Processor	27

PISA	Protocol Independent Switch Architecture	32
IO	Input/Output	6
IP	Internet Protocol	29
SFC	Service Function Chaining	2
CICD	Continous Integration / Continous Deployment	48
BNG	Border Network Gateway	22
MPTCP	Multipath TCP	22
PBLB	Packet-based Load balancing	
FBLB	Flow-based Load balancing	
LCP	Link Control Protocol	24
VIF	Virtual Interface	39
NVM	Node VM	39
EB	Experiment Bridge	39
AVM	Auxiliary VM	39
EVM	Experiment VM	39
MVM	Management VM	39
MB	Management Bridge	39
VITO	VIrtual Testbed Orchestrator	37
PW	Physical Workspace	49
VW	Virtual Workspace	49
IS	Infrastructure Services	49
SVM	Student VM	52
IVM	Infrastructure VM	52
INI	Infrastructure Network Interface	54
TBF	Token Bucket Filter	42

Bibliography and References

Bibliography of the Author

— Conference Papers —

[1] A. Stockmayer, M. Schmidt, and M. Menth, "jLISP: An Open, Modular, and Extensible Java-Based LISP Implementation," in *International Teletraffic Congress*, Würzburg, Germany, Sep. 2016.

[2] A. Stockmayer, C. Kindermann, and M. Menth, "VITO: VIrtual Testbed Orchestration for Automation of Networking Experiments," in *Value Tools*, Venice, Italy, Dec. 2017.

[3] B. Germann, M. Schmidt, A. Stockmayer, and M. Menth, "OFFWall: A Static OpenFlow-Based Firewall Bypass," in *11th DFN-Forum Kommunikationstechnologien*, June 2018.

[4] M. Schmidt, A. Stockmayer, F. Heimgaertner, and M. Menth, "A semi-virtualized testbed cluster with a centralized server for networking education)," in *International Teletraffic Congress*, Sept 2018.

[5] A. Stockmayer, S. Hinselmann, M. Häberle, and M. Menth, "Service function chaining based on segment routing using p4 and sr-iov (p4-sfc)," in *High Performance Computing*, H. Jagode, H. Anzt, G. Juckeland, and H. Ltaief, Eds. Cham: Springer International Publishing, 2020, pp. 297–309.

— Others —

[6] M. Menth, A. Stockmayer, and M. Schmidt, "LISP Hybrid Acces," draft-menth-lisp-ha-00, Internet Engineering Task Force, Jan. 2015. [Online]. Available: https://tools.ietf.org/id/draft-menth-lisp-ha-00.txt

[7] Y. Qu, A. Cabellos-Aparicio, R. Moskowitz, B. Liu, and A. Stockmayer, "Gap Analysis for Identity Enabled Networks," Internet Engineering Task Force, Internet-Draft draft-xyz-ideas-gap-analysis-00, Jul. 2017. [Online]. Available: https://datatracker.ietf.org/doc/draft-xyz-ideas-gap-analysis/00/

General References

[8] VMWare, “VMWare VMotion,” https://www.vmware.com/products/vsphere/vmotion.html.

[9] Red Hat, “libvirt: The Virtualization API,” http://libvirt.org, 2012.

[10] The Linux Foundation, “Xen Project,” https://xenproject.org/.

[11] R. Russell, “virtio: towards a de-facto standard for virtual I/O devices,” *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel*, vol. 42, no. 5, pp. 95–103, 2008.

[12] M. Jones, “Virtio: An I/O virtualization framework for Linux,” https://www.ibm.com/developerworks/library/l-virtio/, 2010.

[13] B. McLellan, “KVM Virtio network performance,” http://blog.loftninjas.org/2008/10/22/kvm-virtio-network-performance/, 2008.

[14] T. linux Kernel team, “Linux control groups,” http://man7.org/linux/man-pages/man7/cgroups.7.html.

[15] T. linux Kernel team, “Network Name Spaces,” http://man7.org/linux/man-pages/man8/ip-netns.8.html.

[16] Docker Inc., “Docker,” https://www.docker.com/.

[17] D. Lezcano, S. Hallyn, S. Graber *et al.*, “Linux Containers,” https://linuxcontainers.org/, 2008.

[18] Intel Corp., “Intel Virtualization Technology (VT-x),” http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html, 2006.

[19] Intel Corp., “APIC Virtualization Performance Testing and Iozone,” https://software.intel.com/en-us/blogs/2013/12/17/apic-virtualization-performance-testing-and-iozone, 2013.

[20] Intel Corp., “Technology Brief: Intel Microarchitecture Nehalem Virtualization Technology,” ttp://download.intel.com/business/resources/briefs/xeon5500/xeon_5500_virtualization.pdf, 2009.

[21] VMware, Inc., “Performance Evaluation of Intel EPT Hardware Assist,” https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf, 2008-2009.

[22] Intel Corp., “Intel Virtualization Technology for Directed I/O (VT-d) Architecture Specification,” http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/vt-directed-io-spec.html, 2012.

[23] PCI SIG, “Root Complex Integrated Endpoints and IOV Updates,” http://pcisig.com/sites/default/files/specification_documents/ECN__Integrated_Endpoints_and_IOV_updates__19%20Nov%202015_Final.pdf, 2015.

[24] Intel Corp., “Intel Virtualization Technology for Connectivity (VT-c),” http://www.intel.com/content/www/us/en/network-adapters/virtualization.html, 2012.

[25] Intel LAN Access Division, “Intel VMDq Technology,” Intel Whitepaper, Intel Corp., Whitepaper, 2008.

[26] PCI SIG, “Single Root I/O Virtualization and Sharing Specification 1.1,” http://www.pcisig.com/specifications/iov/single_root/, 2010.

[27] A. Kivity *et al.*, “kvm: the Linux Virtual Machine Monitor,” in *Linux Symposium*, vol. 1, 2007, pp. 225–230.

[28] F. Bellard and QEMU team, “QEMU – the FAST! processor emulator,” http://wiki.qemu.org/ChangeLog/2.8, 2016.

[29] E. Rosen, A. Viswanathan, and R. Callon, “Multiprotocol label switching architecture,” Internet Requests for Comments, RFC Editor, RFC 3031, January 2001, http://www.rfc-editor.org/rfc/rfc3031.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3031.txt

[30] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, “The locator/id separation protocol (lisp),” Internet Requests for Comments, RFC Editor, RFC 6830, January 2013, http://www.rfc-editor.org/rfc/rfc6830.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6830.txt

[31] N. Leymann, B. Decraene, C. Filsfils, M. Konstantynowicz, and D. Steinberg, “Seamless mpls architecture,” Working Draft, IETF Secretariat, Internet-Draft draft-ietf-mpls-seamless-mpls-07, June 2014, http://www.ietf.org/internet-drafts/draft-ietf-mpls-seamless-mpls-07.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-mpls-seamless-mpls-07.txt

[32] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, “Tcp extensions for multipath operation with multiple addresses,” Internet Requests for Comments, RFC Editor, RFC 6824, January 2013, http://www.rfc-editor.org/rfc/rfc6824.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6824.txt

[33] N. Leymann, C. Heidemann, M. Zhang, B. Sarikaya, and M. Cullen, “Huawei’s gre tunnel bonding protocol,” Internet Requests for Comments, RFC Editor, RFC 8157, May 2017.

[34] M. Menth, A. Stockmayer, and M. Schmidt, “Lisp hybrid access,” Working Draft, IETF Secretariat, Internet-Draft draft-menth-lisp-ha-00, July 2015, http://www.ietf.org/internet-drafts/draft-menth-lisp-ha-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-menth-lisp-ha-00.txt

[35] X. Wei, C. Xiong, and Ed, “Mptcp proxy mechanisms,” Working Draft, IETF Secretariat, Internet-Draft draft-wei-mptcp-proxy-mechanism-02, June 2015, http://www.ietf.org/internet-drafts/draft-wei-mptcp-proxy-mechanism-02.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-wei-mptcp-proxy-mechanism-02.txt

[36] S. Prabhavat, H. Nishiyama, N. Ansari, and N. Kato, "On Load Distribution over Multipath Networks," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 3, pp. 662 – 680, 2012.

[37] K. Sklower, B. Lloyd, G. McGregor, D. Carr, and T. Coradetti, "The ppp multilink protocol (mp)," Internet Requests for Comments, RFC Editor, RFC 1990, August 1996.

[38] *802.1AX: Link Aggregation*, LAN/MAN Standards Committee of the IEEE Computer Society, 2014.

[39] H. Adiseshu, G. Parulkar, and G. Varghese, "A reliable and scalable striping protocol," *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 4, pp. 131–141, Aug. 1996. [Online]. Available: http://doi.acm.org/10.1145/248157.248169

[40] T. Li, D. Farinacci, S. P. Hanks, D. Meyer, and P. S. Traina, "Generic Routing Encapsulation (GRE)," RFC 2784, Mar. 2000. [Online]. Available: https://rfc-editor.org/rfc/rfc2784.txt

[41] J. Heinanen and R. Guerin, "A single rate three color marker," Internet Requests for Comments, RFC Editor, RFC 2697, September 1999.

[42] R. Martin, M. Menth, and M. Hemmkeppler, "Accuracy and Dynamics of Hash-Based Load Balancing Algorithms for Multipath Internet Routing," in *IEEE International Conference on Broadband Communication, Networks, and Systems (BROADNETS)*, San Jose, CA, USA, Oct. 2006.

[43] R. Martin, M. Menth, and M. Hemmkeppler, "Accuracy and Dynamics of Multi-Stage Load Balancing for Multipath Internet Routing," in *IEEE International Conference on Communications (ICC)*, Glasgow, Scotland, UK, Jun. 2007.

[44] R. Stewart, "Stream control transmission protocol," Internet Requests for Comments, RFC Editor, RFC 4960, September 2007, http://www.rfc-editor.org/rfc/rfc4960.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4960.txt

[45] P. P. D. Amer, M. Becke, T. Dreibholz, N. Ekiz, J. Iyengar, P. Natarajan, R. R. Stewart, and M. Tüxen, "Load Sharing for the Stream Control Transmission Protocol (SCTP)," Internet Engineering Task Force, Internet-Draft draft-tuexen-tsvwg-sctp-multipath-17, Jan. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-tuexen-tsvwg-sctp-multipath-17

[46] T. Viernickel, A. Froemmgen, A. Rizk, B. Koldehofe, and R. Steinmetz, "Multipath quic: A deployable multipath transport protocol," in *2018 IEEE International Conference on Communications (ICC)*, May 2018, pp. 1–7.

[47] J. D. Case, M. Fedor, M. L. Schoffstall, and J. R. Davin, "Simple network management protocol (snmp)," Internet Requests for Comments, RFC Editor, STD 15, May 1990, http://www.rfc-editor.org/rfc/rfc1157.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1157.txt

[48] W. Braun and M. Menth, "Software-defined networking using openflow: Protocols, applications and architectural design choices," *Future Internet*, vol. 6, no. 2, pp. 302–336, 2014.

[49] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-Defined Networking: A Comprehensive Survey,” http://arxiv.org/abs/1406.0440, 2014.

[50] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: A Network Programming Language,” in *In Proceedings of the ACM SIGPLAN International Conference on Functional Programming.* Tokyo, Japan, Sep. 2011.

[51] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, “Composing Software-Defined Networks,” in *In Proceedings of the USENIX Syposium on Networked Systems Design & Implementation (NSDI).* Lombard, USA, 2013, pp. 1–14.

[52] A. Voellmy, H. Kim, and N. Feamster, “Procera: A Language for High-Level Reactive Network Control,” in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN).* Helsinki, Finland, Oct. 2012, pp. 43–48.

[53] F. M. Facca, E. Salvadori, H. Karl, D. R. Lopez, P. A. A. Gutierrez, D. Kostic, and R. Riggio, “NetIDE: First Steps towards an Integrated Development Environment for Portable Network Apps,” in *European Workshop on Software Defined Networks (EWSDN).* Berlin, Germany, 2013, pp. 105–110.

[54] Open Networking Foundation members, “OpenFlow Switch Specification,” The Open Networking Foundation.

[55] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, “Network Configuration Protocol (NETCONF),” RFC 6241 (Proposed Standard), Internet Engineering Task Force, Jun. 2011. [Online]. Available: https://tools.ietf.org/html/rfc6241.txt

[56] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and et al., “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[57] Cisco and/or its affiliates, “OpFlex: An Open Policy Protocol,” Cisco Whitepaper, Cisco, Whitepaper, 2014.

[58] M. Smith, R. Adams, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher, “OpFlex Control Protocol,” Internet-Draft (Informational), Internet Engineering Task Force, Nov. 2014. [Online]. Available: https://tools.ietf.org/html/draft-smith-opflex-01

[59] M. Bjorklund, “YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF),” RFC 6020 (Proposed Standard), Internet Engineering Task Force, Oct. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc6020.txt

[60] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, “Forwarding and Control Element Separation (ForCES) Protocol Specification,” RFC 5810

(Proposed Standard), Internet Engineering Task Force, Mar. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5810.txt

[61] L. Yang, R. Dantu, T. Anderson, and R. Gopal, "Forwarding and Control Element Separation (ForCES) Framework," RFC 3746 (Informational), Internet Engineering Task Force, Apr. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3746.txt

[62] T. V. Lakshman, T. Nandagopal, R. Ramjee, K. Sabnani, and T. Woo, "The SoftRouter Architecture," in *In Proceedings of the ACM HotNets*. San Diego, USA, Nov. 2004.

[63] P. Quinn and T. Nadeau, "Problem statement for service function chaining," Internet Requests for Comments, RFC Editor, RFC 7498, April 2015, http://www.rfc-editor.org/rfc/rfc7498.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7498.txt

[64] P. Quinn, U. Elzur, and C. Pignataro, "Network service header (nsh)," Internet Requests for Comments, RFC Editor, RFC 8300, January 2018.

[65] A. Farrel, S. Bryant, and J. Drake, "An mpls-based forwarding plane for service function chaining," Internet Requests for Comments, RFC Editor, RFC 8595, June 2019.

[66] F. Clad, X. Xu, C. Filsfils, D. Bernier, C. Li, B. Decraene, S. Ma, C. Yadlapalli, W. Henderickx, and S. Salsano, "Service programming with segment routing," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-spring-sr-service-programming-05, September 2021, https://www.ietf.org/archive/id/draft-ietf-spring-sr-service-programming-05.txt. [Online]. Available: https://www.ietf.org/archive/id/draft-ietf-spring-sr-service-programming-05.txt

[67] ETSI, "ETSI Mano," https://www.etsi.org/technologies/open-source-mano.

[68] T. L. Foundation, "OPNFV," https://www.opnfv.org.

[69] Amazon, "tcpdump," https://aws.amazon.com.

[70] Microsoft, "Microsoft azure," https://azure.microsoft.com/.

[71] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 71–84. [Online]. Available: https://doi.org/10.1145/3098822.3098828

[72] A. Mohammadkhan, S. Panda, S. G. Kulkarni, K. K. Ramakrishnan, and L. N. Bhuyan, "P4nfv: P4 enabled nfv systems with smartnics," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2019, pp. 1–7.

[73] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou, "P4sc: Towards high-performance service function chain implementation on the p4-capable device," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 1–9.

[74] T. N. Team, “Direct Code Execution,” https://www.nsnam.org/about/projects/direct-code-execution/.

[75] Tetcos, “NetSim,” https://www.tetcos.com/.

[76] B. Lantz, B. Heller, and N. McKeown, “A Network in a Laptop: Rapid Prototyping for Software-defined Networks,” in *ACM HotNets*, 2010.

[77] IEEE Computer Society, “IEEE 802.1D Standard,” http://standards.ieee.org/getieee802/download/802.1D-1998.pdf, 2004.

[78] Simon Kelley, “dnsmasq,” http://www.thekelleys.org.uk/dnsmasq/doc.html, 2017.

[79] G. Kroah-Hartman, “udev – A Userspace Implementation of devfs,” in *Linux Symposium*, 2003.

[80] S. Shepler *et al.*, “Network File System (NFS) version 4 Protocol,” RFC 3530, Apr. 2003.

[81] Simson L. Garfinkel, “tcpflow,” https://github.com/simsong/tcpflow, 2017.

[82] tcpdump team, “tcpdump,” http://www.tcpdump.org/.

[83] QEMU Team, “qcow2,” http://git.qemu-project.org, 2017.

[84] The Linux foundation, “tso,” https://wiki.linuxfoundation.org/networking/gso, 2017.

[85] A. Kuznetsov and S. Hemminger, “iproute2: Utilities for Controlling TCP/IP Networking and Traffic,” 2012.

[86] The Linux foundation, “netem,” https://wiki.linuxfoundation.org/networking/netem, 2017.

[87] The Linux foundation, “tbf,” https://linux.die.net/man/8/tc-tbf, 2017.

[88] S. Floyd and V. Jacobson, “Random Early Detection Gateways for Congestion Avoidance,” *IEEE/ACM Trans. Netw.*, Aug. 1993.

[89] Daniel Stenberg , “cURL,” https://curl.haxx.se/, 2017.

[90] busybox developer team, “BusyBox,” https://busybox.net/, 2015.

[91] K. Kawaguchi, “Jenkins,” https://www.jenkins.io, 2020.

[92] G. Inc., “Gitlab CI,” https://docs.gitlab.com/ee/ci/.

[93] J. Liebeherr and M. E. Zarki, *Mastering Networks – An Internet Lab Manual.* Pearson Education, 2003.

[94] M. Schmidt, F. Heimgaertner, and M. Menth, “Demo: A Virtualized Lab Testbed with Physical Network Outlets for Hands-on Computer Networking Education,” in *ACM SIGCOMM*, 2014.

[95] M. Schmidt, F. Heimgaertner, M. Hoefling, and M. Menth, “A Virtualized Testbed with Physical Outlets for Hands-on Computer Networking Education,” in *ACM SIGITE*, 2014.

[96] *IEEE 802.1Q: Virtual Bridged Local Area Networks*, LAN/MAN Standards Committee of the IEEE Computer Society, 2003.

[97] J. Sermersheim, "Lightweight Directory Access Protocol (LDAP): The Protocol," RFC 4511 (Proposed Standard), Internet Engineering Task Force, Jun. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4511.txt

[98] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Network File System (NFS) version 4 Protocol," RFC 3530 (Proposed Standard), Internet Engineering Task Force, Apr. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3530.txt

[99] L. Kanies, "Puppet: Next-Generation Configuration Management," *The USENIX Magazine*, vol. 31, no. 1, pp. 19–25, 2006.

[100] Theodore Ts'o, "e2fsprogs," https://sourceforge.net/projects/e2fsprogs/, 2016.

[101] Red Hat, "Logical Volume Manager (LVM)," https://sourceware.org/lvm2/, 2016.

[102] J. Case, R. Mundy, D. Partain, and B. Stewart, "Introduction and Applicability Statements for Internet Standard Management Framework," RFC 3410 (Proposed Standard), Internet Engineering Task Force, Dec. 2002. [Online]. Available: https://tools.ietf.org/html/rfc3410.txt

[103] Gentoo Foundation, "Gentoo Linux," http://www.gentoo.org, 2017.

[104] Internet Systems Consortium, "ISC DHCP," https://www.isc.org/downloads/dhcp/.

[105] R. Hawkins *et al.*, "Linux IPv6 Router Advertisement Daemon (radvd)," http://www.litech.org/radvd/, 2016.

[106] Internet Systems Consortium, "BIND," https://www.isc.org/downloads/bind/.

[107] The OpenLDAP project, "openldap," http://www.openldap.org/.

[108] Network Time Foundation, "The Network Time Protocol Project," http://www.ntp.org/.

[109] Sysoev, Igor and Nginx, Inc., "nginx," https://www.nginx.com/.

[110] M. Pahl, "The ilab concept: Making teaching better, at scale," *IEEE Communications Magazine (Commag)*, vol. 55, no. 11, pp. 178–185, 2017.

[111] M. Gutschke *et al.*, "shellinabox," https://github.com/shellinabox/shellinabox.

[112] The LXQt Team, "The Lightweight Qt Desktop Environment," http://lxqt.org/.

[113] B. Gurudoss, P. Jakma, T. Teräs, G. Troxel, and Quagga developer team, "Quagga Routing Suite," http://www.nongnu.org/quagga/.

[114] Red Hat, "SPICE," http://www.spice-space.org/, 2012.

[115] A. Kuznetsov and S. Hemminger, "tc: Show/Change traffic control settings," 2012.

[116] Giuseppe Scrivano and wget developer team, "GNU Wget," https://www.gnu.org/s/wget/.

[117] B. Perens, E. Andersen, R.Landley, and D. Vlassenko, “busybox httpd,” https://busybox.net/.

[118] P. Eardley, “Metering and marking behaviour of pcn-nodetockm,” Internet Requests for Comments, RFC Editor, RFC 5670, November 2009.

[119] iperf3 team, “iperf3,” http://software.es.net/iperf/.

[120] Aaron Wood, “Iperf Microbursts Issue,” http://burntchrome.blogspot.com/2016/09/iperf3-and-microbursts.html, 2016.

[121] *802.1ab: Provider Bridges*, LAN/MAN Standards Committee of the IEEE Computer Society, 2005.