TECHNISCHE UNIVERSITÄT DRESDEN

ComNets
Deutsche Telekom Chair of Communication Networks

Faculty of Electrical and Computer Engineering Institute of Communication Technology

Deutsche Telekom Chair of Communication Networks

Design and Evaluation of a Network Softwarization Platform for Edge and In-Network Computing

Patrick Ziegler
Born on: February 5, 1991 in Bad Soden-Salmünster
Matriculation number: 3750096

Diploma Thesis

to achieve the academic degree

Diplomingenieur (Dipl.Ing.)

Supervisors
Dipl.-Ing. Zuo Xiang
Dr.-Ing. Roland Schingnitz

Supervising professor
Prof. Dr.-Ing. Dr. h.c. Frank H. P. Fitzek

Submitted on: October 10, 2019

TECHNISCHE UNIVERSITÄT DRESDEN
FAKULTÄT ELEKTROTECHNIK UND INFORMATIONSTECHNIK

Topic for the Diploma thesis

For Mr Patrick Ziegler (3750096) ET/2011

Theme Design and Evaluate a Network Softwarization Platform for Edge and In-Network Computing.

Task description

In order to support heterogeneous 5G (and beyond) use cases and their corresponded performance requirements (e.g. minimum throughput and maximal allowed latency), adaptivity ,scalability and programmability play an more important role in next generation of communication networks. To meet the ultra low latency requirements, the emerging paradigm edge computing focuses on bringing the computation as close to the source of data as possible. Due to the limited capabilities and at the edge, it is an open challenge to design, develop and evaluate a lightweight virtualization platform which supports management and orchestration (MANO) of dynamically placed and migrated virtualized network functions (VNFs). The main goal of the thesis is to design, develop, and evaluate an adaptive network softwarization platform for edge and in-network computing, with a focus on reducing end-to-end service latency and supporting flexible VNFs placement.

Target:

- Utilize the emerging and suitable paradigms (e.g. serverless computing), orchestration and data plane technologies to improve the adaptivity and resource-efficiency of the conventional NFV platforms.
- Design flexible placement and migration algorithms that take end-to-end service latency and resource-efficiency into consideration. Both stateful and stateless network functions should be supported.
- Perform a comprehensive performance evaluation of the end-to-end service delay and resource-efficiency (e.g. CPU and memory usage of each physical node) of the design with emulation and practical implementation.

Supervisor: Dipl.-Ing. Zuo Xiang

First Reviewer: Prof. Dr.-Ing. Dr. h. c. Frank Fitzek

Second Reviewer: Dr.-Ing. Roland Schingnitz

Started at: 02.05.2019

To be submitted by: 10.10.2019

The diploma thesis is written in English / ~~German~~.

Prof. Dr.-Ing. Steffen Bernet
Chairman oft he Examination Board
for Electrical Engineering

Prof. Dr.-Ing. Dr. h. c. Frank Fitzek
Responsible Tutor

Declaration of Authorship

I hereby certify that this report has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this report. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Dresden, Thursday 10th October, 2019

Abstract

This work proposes a new architecture for Network Function Virtualization (NFV) platforms following the goal of improving the adaptibility and scalability of state of the art solutions.

An integrated network monitoring solution based on state of the art data plane technologies like Extended Berkleey Packet Filter (eBPF) and eXpress Datapath (XDP) is introduced to make network softwarization platforms aware of the underlay network and provide accurate link latency measurements to be used for modeling the network topology.

Based on this input, the architecture of conventional NFV platforms was modified to support an automatic network service provision based on the serverless paradigm and a new algorithm for latency optimized network service embedding is developed.

Kurzfassung

In der vorliegenden Arbeit wird eine neue Architektur für software-basierte Netzwerklösungen mit dem Ziel der Verbesserung der Anpassungsfähigkeit konventioneller Plattformen entwickelt.

Hierzu wird ein integriertes System zur Überwachung wichtiger Netzwerkparameter wie etwa Link-Latenz u.a. vorgestellt, welches Messdaten für eine umfassende Modellierung der zugrunde liegenden Netzwerktopologie zur Verfügung stellt.

Diese Daten werden in einer speziell angepassten Plattform-Architektur für eine automatische und bedarfsgerechte Bereitstellung von Netzwerkfunktionen eingesetzt. Hierzu wurde ein Algorithmus zur Latenz-Optimierung entwickelt.

Contents

List of Figures

List of Tables

Glossary

ACL	Access Control List
API	Application Programming Interface
AMQP	Advanced Message Queuing Protocol
BSS	Business Support System
CPU	Central Processing Unit
COIN	Computing in the Network
DevOps	Development and Operations
DDoS	Distributed Denial of Service
DPDK	Data Plane Development Toolit
DPI	Deep Packet Inspection
DPID	Data Path ID
eBPF	Extended Berkleey Packet Filter
EMS	Element Management System
ETSI	European Telecommunications Standards Institute
FaaS	Function as a Service
FCAPS	Fault, Configuration, Accounting, Performance and Security
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
ICMP	Internet Control Message Protocol
ID	Identifier
IETF	Internet Engineering Task Force
IoT	Internet of Things

IP	Internet Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KPI	Key Performance Indicator
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
MANO	Management and Orchestration
MEC	Mobile Edge Cloud
ML2	Modular Layer 2
MPLS	Multiprotocol Label Switching
NAT	Network Address Translation
NIC	Network Interface Card
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NFVO	NFV Orchestrator
NFVM	NFV Manager
NSD	Network Service Descriptor
NSH	Network Service Header
NTP	Network Time Protocol
NP	Nondeterministic Polynomial Time
NSH	Network Service Header
OASIS	Organization for the Advancement of Structured Information Standards
ONF	Open Networking Foundation
OOP	Object Oriented Programming
ONOS	Open Network Operating System
OSM	Open Source MANO
OSS	Operations Support System

OvS	Open vSwitch
PID	Process Identifier
PoC	Proof of Concept
QoS	Quality of Service
ReST	Representational State Transfer
RTT	Round Trip Time
RSS	Resident Set Size
SDN	Software Defined Networking
SFC	Service Function Chain
SI	Service Index
SPI	Service Path Identifier
TC	Traffic Control
TCP	Transmission Control Protocol
TLV	Type-Length-Value
TTL	Time-to-Live
TOSCA	Topology and Orchestration Specification for Cloud Applications
UUID	Universally Unique Identifier
VLAN	Virtual Local Area Network
VIM	Virtualized Infrastructure Management
VNF	Virtual Network Function
VNFM	VNF Manager
XDP	eXpress Datapath

1 Introduction

1.1 Context

Recent advances in improving throughput and minimizing latency in modern communication networks heavily rely on the realization of necessary functionality as highly optimized and integrated hardware components.

While individually providing a very efficient solution to the problem at hand, this approach has the drawback to be very inflexible and may lead to suboptimal network parameters in volatile environments. Having fixed building blocks in a network deployment furthermore implies high cost of future changes such as adding functionality to an already deployed setup.

This could be a problem for future 5G network environments, which are envisioned to be more than just another incremental improvement of network parameters and are expected to support an extremely wide range of applications in various fields of communication [7].

Use Cases

One of these is the communication between humans and machines as envisioned in current research for the tactile internet, where humans remotly control a robot or some other process and therefore latency is of utmost importance. Remote surgery and augmented reality are examples for appliances of this concept.

But also communication between machines is a field where 5G networks are expeted to be of great use. Especially in context of industry automation and the concept of smart cities is this 5G a hot topic. Here, not only latency restrictions, but also resilience and adaptibility considerations could impose requirements on future network solutions.

And as a dependency for the two aforementioned fields of communication, sensor

networks for modeling and control and Internet of Things (IoT) applications could provide humans as well as machines with useful information.

Applications in those fields may demand an individual set of services with heterogeneous requirements which all have to be provided by the same network environment.

As the set of services is likely to change over time (due to user requests or changing environments) the trend of providing services as hardware based solutions is currently reversing itself and strategies for network softwarization and virtualization are under active research [8].

Network Function Virtualization

Using Virtual Network Functions (VNFs) to decouple services from hardware components would have the benefit of enabling communication platforms to dynamically adapt to changing user requests and environments as the necessary services are provided by virtualized software components that run on standard commodity hardware and can be scaled and migrated as the need arises.

This concept is called Network Function Virtualization (NFV) and is currently not only under active research, but also in the beginning of standardization whereby the latter is mainly carried out by the ETSI that begun its work on NFV in 2012.

Edge Computing

Most of the time, NFV is only thought of as a fancy way to provide traditional network services like Firewalls, Deep Packet Inspection (DPI) or Network Address Translation (NAT) in a very flexible way. But having the necessary infrastructure for dynamically deploying various functions in the network naturally leads to the idea of having arbitrary services available at the network edge.

This concept is commonly referred to as Mobile Edge Cloud (MEC), Edge Computing or In-Network Computing and two things make this approach unique.

Firstly, the *Computing* in those terms refers to arbitrary functionality that does not neccessarily has to provide network related services but rather general facilities such as machine learning or other tasks, that are either to complex for a customer to solve by itself (as often the case in IoT) or require more than one endpoint to

cooperate in a distributed task.

Secondly, referring to an *Edge* in the network implies the placement of computing facilities as close to the end user as possible.

Such a system would have the benefit of potentally reducing the neccessary network bandwith for some tasks and improved latency for others whereby the first case obviousley occurs on data compressing tasks like object detection or transcoding and the second case could have a huge impact on augmented reality and remote controlling applications.

Recently the term *Fog Computing* was introduced. Fog and Edge Computing differ from each other in the placment of a function in the network. Whereas the latter refers to having a function deployed on the first gateway to which an end user is connected to, the term Fog Computing describes functionality that is deployed *somewhere close* to that endpoint. Either way, both terms relate to a deployment that is latency aware (as latency usually constitutes the *distance* in the network) unlike the current state of the art of Cloud Computing, where all the computing facilities are deployed on centralized cloud datacenters and cannot inherently optimize for the aforementioned use cases.

As the author regards Edge Computing as an abstraction or an extended usage of Network Function Virtualization (NFV), only NFV will be used as a term to refer to virtualized computing facilites in the network for the rest of this work.

2 State of the Art

2.1 Software Defined Networking (SDN)

Even though providing virtualized network functionality instead of specialized hardware solutions could also be applied in traditional networks, the added value of a dynamic management of VNFs can most effectively be utilized with SDN providing the deployment context. Therefore, the author considers NFV and SDN to be complementary concepts.

To the best of the authors knowledge, there is currently no network softwarization platform available, that is not ryling on SDN for network management. Therefore, in order to evaluate the state of the art of NFV, it is necessary to have profound knowledge about SDN as it is used in the context of NFV.

2.1.1 Concept

The core idea of SDN is to decouple the configuration and management of network elements from the underlying datapath operations, in order to eliminate the need for manual configuration of network elements and introduce a concept called *programmable datapath*.

This term describes the fact that with establishing standardized interfaces between well defined components of a layered controlling architecture, SDN allows network operators to implement software components that describe arbitrary forwarding policies to be executed in the network in a very similar way a programmer would write software for executing arbitrary tasks on standard computing hardware.

In this analogy, the instruction set of a Central Processing Unit (CPU) together with operating system functions and a standard library would serve the same purpose the components of SDN do, that is to provide an abstracted view of the underlying hardware components to make the development of complex tasks easyer (or even

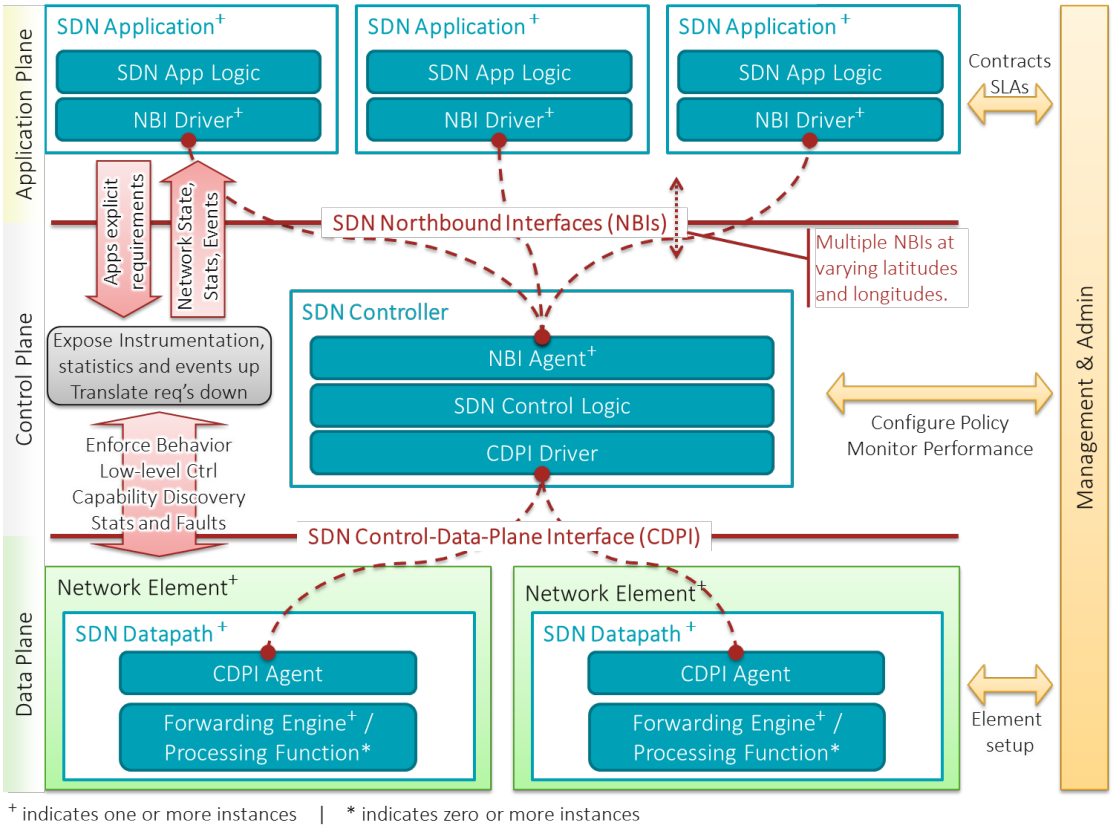

Figure 2.1: The SDN reference architecture is providing a layered view of the network that allows to dynamically implement and deploy arbitrary forwarding policies (Image source: [1])

possible). In that sense, SDN serves as an operating system for the network.

The components that make up the SDN architecture are visualized in figure 2.1. As the Open Networking Foundation (ONF) suggested in their whitepaper [9] in 2012, the architecture is layered into data, control and application plane.

All network elements that together constitute the networks datapath of belong to the data plane as the lowest layer of this hierarchy. In addition to the forwarding engine, they provide functionality for analyzing and matching network traffic as well as routines for manipulating individual data packets. Hardware vendors can furthermore define spezialized functionality on their own as will be shown in the next section. Most SDN enabled hardware comes with a monitoring that is able to collect various statistics on the network traffic and its operation.

The forwarding and processing engines of network elements are configured by SDN controllers that naturally belong to the control plane. Network elements may be connected to more than one controller as they are coupled in a many-to-many relationship. The controllers usually keep track of their associated peers and have broader view of the network. When a network element encounters packets that do not match any rule in the forwarding or processing engines, the packet will be usually sent to the SDN contoller who has to ultimately decide what to do with it, and this decision may possibly be derived from information he has previously gathered from other network elements).

But strictly speaking, such decisions are not made by the controller itself, but rather by SDN applications in the aforementioned application layer. They are responsible for mapping user requests and the network state information collected by the controller either reactively (as events occur) or proactively to changes in datapath configuration or individual actions.

With the outsourcing of application logic to a higher layer, the SDN controller mainly serves as a gateway for its north- and southbound interfaces. The term *northbound* describes the interfaces to SDN applications on a higher layer and the term *southbound* describes interfaces to network elements on a lower layer.

2.1.2 Protocol Specification

Whereas the term SDN is describing the concept of decoupling datapath functions and configuration management, the concrete design of all involved components and interfaces is left to protocols implementing the SDN concept.

An example of such a protocol is OpenFlow, which is actively developed by the ONF in form of detailed descriptions of interfaces between SDN controllers and switches and their desired behaviour. The latest release can be found in the *OpenFlow Switch Specification Version 1.5.1* [10] that was published in 2015.

To properly serve as a protocol for SDN, OpenFlow defines as set of messages that are usually exchanged between controllers and switches over the Transmission Control Protocol (TCP). An overview of commonly seen messages is given in table 2.1. These messages allow the collection of per-flow statistics as well as the configuration of all components of an OpenFlow switch.

A simplified view of the architecture of such switch is shown in figure 2.2. Instead of routing tables as known of traditional networking hardware, where the next hop

Direction	Message name	Description
Controller-to-Switch	FeatureRequest	Requesting Information on switches abilities to collect stats
	FlowMod, GroupMod etc.	Modify (ADD, DELETE, MOD) the given component
	StatsRequest	Request statistics of Flows, Ports etc. (specified via flags)
	PacketOut	Send the given packet (either from buffer or payload) out on the specified port
Symm-etric	Hello	Beginning of peering sequence
	Echo	Requesting EchoReply Message from peer to check vital signs or latency
Switch-to-Controller	PacketIn	Packet encouncered with no matching flow table entry
	FlowRemoval	Flow table entry removed due to timeout
	Error	An Error occured

Table 2.1: Commonly used OpenFlow Messages sorted by Direction

for a given destination address is looked up via Longest Prefix Matching, SDN enabled switches contain *Flow Tables* that are configured by SDN controllers via the OpenFlow control channel.

Flow tables are the main part of an OpenFlow switch as they provide a mapping of actions on incoming traffic based on some filter criteria called *Classifiers*. Classifiers for incoming packets are specified as attributes of *Match* structures which are part of every entry in a flow table. Such attributes typically relate to header fields of common network protocols such as Ethernet and Internet Protocol (IP) but can also require metadata such as the ingress port number of incomping packets to be of a certain value to match its entry in the flow table.

An incoming packet is matched to every flow table entry until the first hit occurs. Flow tables entrys are sorted by their priority value. Therefore, when multiple entries share a subset of classifiers, the entry with the highest priority value will always be selected. If a matching flow entry is found, the associated instructions are carried out. An instruction is an operation to be executed by the switch, such as forwarding the packet to some egress port or to apply a set of actions on it. An action is an operation to be executed on the given packet, such as modifying header

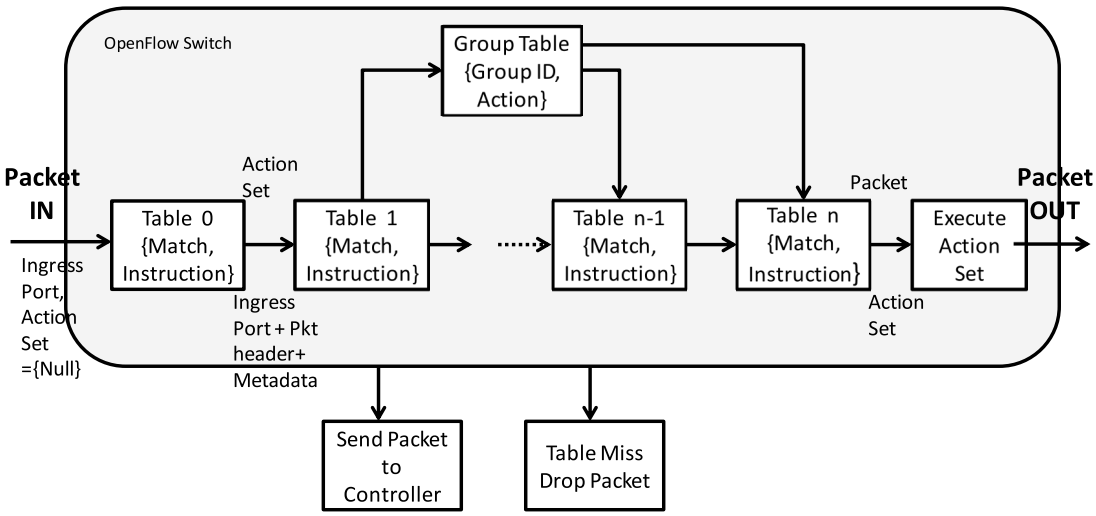

Figure 2.2: OpenFlow switches can provide multiple flow tables containing possibly multiple instructions (or instruction groups) that specify how to handle matching packets (Image source: [2])

values or carrying out Virtual Local Area Network (VLAN) encapsulation.

Hardware vendors are allowed to implement custom actions that can be applied via the so called *Experimenter Action* which is mapped to the given extentions via an Identifier (ID). This mechanism makes it possible to add custom functionality without having to define new OpenFlow action types. But as only vendors have the ability to add functionality to their hardware, it is not possible for users and SDN developers to introduce use case specific extensions via the experimenter mechanism.

Multiple flow tables may exist on a switch. The process of matching always starts with the root flow table and other tables will only be evaluated when the appropriate forwarding instructions were used.

Incoming packets are not only allowed to be forwarded to other flow tables or ports, but also to so called *Group Tables*. These differ from flow tables as they have no matching mechanism implemented but only allow packet modification and forwarding. Group table entries are divided into collections of actions called *buckets* who are handled depending on the group type.

Most notably, the group type SELECT can be used for load balancing as every bucket in the group has a *weight* parameter assigned to. Packets sent to the group are then handled by one of the defined buckets while proportional amount of traffic directed

to a bucket is represented by the bucket weight divided by the sum of all weights.

Another important use case of group tables is to provide alternative routing decisions when the originally choosed path is not available any more. This functionality is provided by the group type FAST_FAILOVER for which each defined bucket represents an alternative route if its predecessor is for some reason not available.

Besde the aformentioned applications, groups are generally used for operations that have to be applied to multiple flow entrys to prevent the need for keeping track individual flow entries which simplifies the process of updating the configuration.

2.1.3 Virtualized Networking Infrastructure

With the introduction of software based management and control logic in SDN, also the deployment of network entities themselves is increasingly becoming independent of physical hardware.

Even though dedicated harware switches undoubtedly play a very important role for network speed—and even more so in the context of SDN where flow handling is much more complex than routing via longest prefix matching as implemented in traditional network hardware—the recent trend for ubiquitous virtualization led to the development software based SDN switches. This is a manifestation of the fact that differences between hardware and software solutions are increasingly becoming indistinct as the layer of abstraction introduced by SDN allows the deployment of appropriate solutions as needed.

Probably the most widely-used software implementation of a SDN enabled switch is Open vSwitch (OvS), which finds its application as the backbone of major SDN and NFV platforms like OpenStack and others. Its architecture is shown in figure 2.3. The involved software components are decoupled from the emulated switch instances and subdivided in a database server and a switching daemon in userspace and several kernel modules providing the in-kernel *fast path*.

The user space daemon ovs-vswitchd is an implementation of the OpenFlow protocol and manages the actual forwarding logic implemented as in-kernel datapath [3]. Routing decisions previously derived by matching flow table entries in ovs-vswitchd are subsequently cached for faster operation in the in-kernel fast path [11].

The term *fast path* indicates the presence of a *slow path* in the data plane of OvS. The latter route is taken by packets belonging to a flow specification that was not

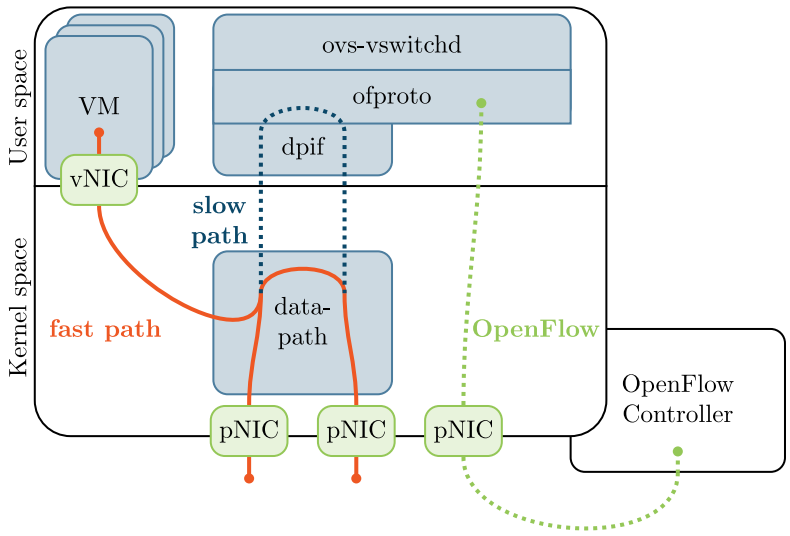

Figure 2.3: Open vSwitch (OvS) is composed of a user space daemon implementing the OpenFlow protocol and several kernel modules implementing the in-kernel *fast path* (Image source: [3])

cached already. Such packets are using the the slow path via the switching daemon to be mapped to a flow entry in the switches database whereafter the associated instructions are executed and the flow information is cached in kernel space. Subsequent packets matching the same flow are then directly processed in kernel space with increased speed.

In that regard, OvS is based on a similar concept as SDN itself, as the switching daemon serves as a kind of intermediate controller and routing decisions are cached in kernel space (and may be revoked after some time) as the need arises and requests to the switching daemon are issued when unkown packets are entering the in-kernel fastpath.

But in order to make the *slow path* a little less slow, a considerable amount of work has been invested in researching ways to avoid unneccessary processing in the kernel network stack or costly context switches. This has led to the development of ovs-dpdk that is incorporating the Data Plane Development Toolit (DPDK) for directly handling incoming packets in user space.

With the developtment of XDP in recent Linux kernels (as will be described in section 2.5), the work on accelerating the OvS slowpath is currently shifting towards XDP based solutions as this approach promises to be most convenient for deployment and shows very high processing rates [12].

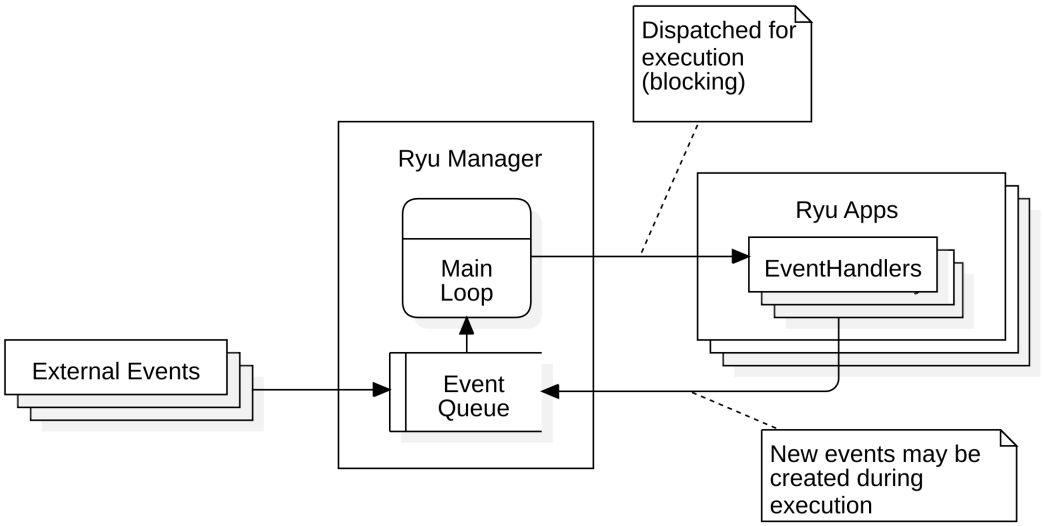

Figure 2.4: Execution model of the Ryu event loop

The authors of [12] are motivating their work on XDP with the prominent role OvS plays in the context of NFV where it is deployed to switch traffic between different VNFs and the minimization switching overhead is of great importance. This work will elaborate the outstanding role of OvS and XDP in a different aspect of NFV in the follwing chapters.

2.1.4 Controllers and Network Operating Systems

A review of Software Defined Networking (SDN) is not complete without having a closer look on the controller and its associated applications as these components comprise an important part of the SDN concept.

Ryu Network Operating System

A most prominent example of a network operating system often used for research projects on the field of NFV is the Ryu SDN Framework and its associated controller architecture. Custom Ryu applications are implemented in Python and can incorporate the built-in Ryu library for handling packets with a wide range of supported protocols such as Ethernet, IP, VLAN and others.

The execution model of Ryu is shown in figure 2.4. With its modularized structure,

it is honoring the layered composition of the SDN architecture shown above. The core part of Ryu is its multithreaded execution stage consisting of an (infinite) loop constantly supervising the event queue where all occuring events are temporarily stored. Such events could be created by the associated TCP server on incoming OpenFlow messages. This server is implementing the interface to the associated network elements and should usually be listening on TCP port 6653 (or 6633 in earlier releases of the standard) as specified in [10].

Developing applications on top of the Ryu SDN controller is done in a way generally used in Object Oriented Programming (OOP). An application is represented by an object whose class definition is inheriting its core from an universal application base class provided by Ryu. Via the inherited functions for registering event handlers, the newly defined applications are able to register some of their functions as event handlers for various types of events. These could be any of the events described in section 2.1.2.

If an event occurs, it is forwarded to all associated event handlers by the Ryu main loop. During the execution of a previously enqueued event, more events could be created which in turn will be enqueued for execution into the main event queue.

With this highly parallelized architecture where individual SDN applications are independent of the underlying SDN logic, Ryu is exemplary honoring the concept of SDN and allows the development of distributed and automatically scaled SDN controllers.

Open Network Operating System (ONOS)

A more elaborated example of a network operating system is Open Network Operating System (ONOS). That is an industy-ready implementation of a distributed SDN controller providing first class fault tolerance and comfortable configuration management via a graphical web interface.

Unlike Ryu, which is can also be incorporated by SDN platforms like OpenStack but is in gerneral a standalone product, ONOS has built-in support for OpenStacks Modular Layer 2 (ML2) interface specification (more on that below) and can therefore serve as an alternative to OpenStacks default SDN implementation contained in the Neutron (the networking module of OpenStack). This feature can be used to improve the awareness of OpenStack for the abstracted underlay network which in turn could be necessary for advanced applications like the network monitoring sys-

tem proposed in this work. But to the best of the authors knowledge, no research attempts in that direction have been published yet.

2.2 Network Function Virtualization (NFV)

After having provided the reader with an overview of SDN, the following sections contain a discussion of NFV as an applicaton on top of the aforementioned concepts. It is common practice among NFV developers and research groups to map developed or investigated components to the reference architecture described in section 2.2.1.

This reference model not only serves as the basis of a future standard on this field, but also provides a common set of terms for typical roles and components of NFV platforms and by that facilities the cooperation between researchers as well as the exchange of ideas and consequently helps to consolidate the concept of NFV and its development.

Development and standardization of NFV is mainly organized by the European Telecommunications Standards Institute (ETSI). This group started its work on NFV in 2012 and is organizing all publications (whitepapers, standardization efforts etc.) in release cycles of about two years.

Very recently, the beginning of the fourth release cycle of standardization efforts was announced especially formulating the following goals concerning Management and Orchestration (MANO) [13].

- Decouple VNFs from virtualized Infrastructure and support lightweight virtualization
- Simplify management and orchestration of VNFs by improving life-cycle management and supporting autonomous networking
- Simplify development and deployment of NFV by improving MANO capabitilies

Compared to previous releases, a shift away from merely providing NFV capabitilies towards improving the management of VNFs can be observed. This coincides with the research question of this work, but to the best of the author knowledge and apart from the formulation of directions as stated above, no further work has been published by ETSI yet.

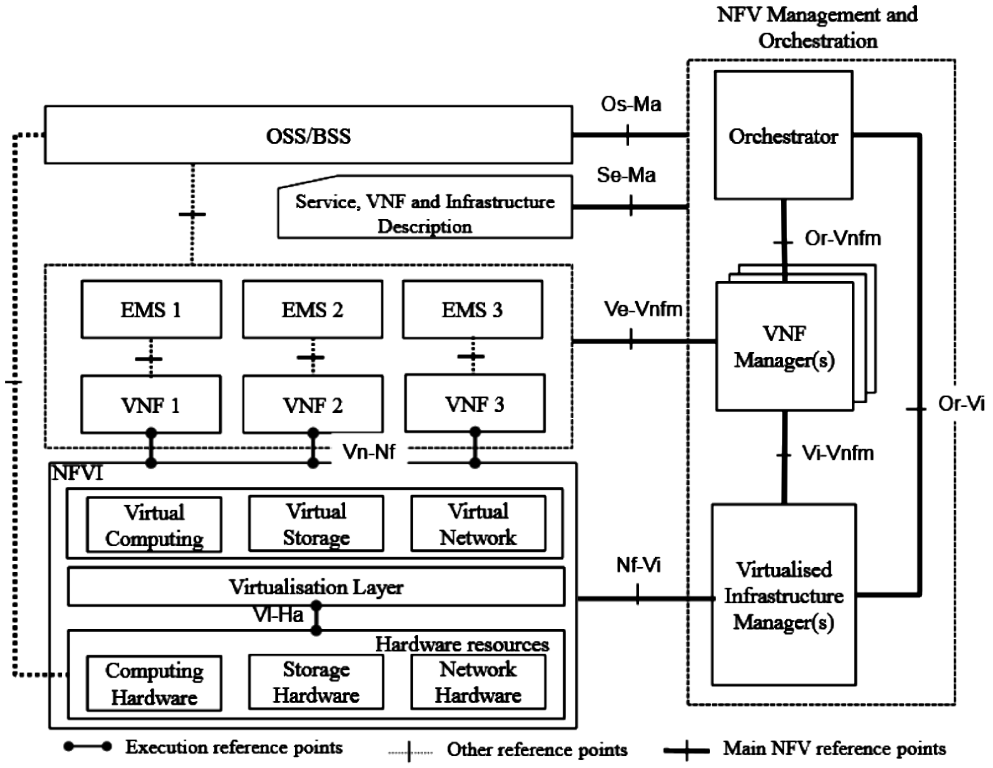

Figure 2.5: Network Function Virtualization (NFV) reference architecture as proposed by the European Telecommunications Standards Institute (ETSI) (Image source: [4])

2.2.1 Reference Architecture

An important result of early standardization efforts by ETSI is the reference architecture depited in figure 2.5. This is an overview of all envisioned components and interfaces of NFV platforms.

The interface to users and operators of the network is provided Operations Support System (OSS) and Business Support System (BSS). These services allow transactions like registering network services and other more business related tasks like billing and authentication to be forwarded to the MANO framework.

Network services are requested via providing the framework with a formalized description of the individual VNFs and their interconnection. Such a description is called Network Service Descriptor (NSD) and an example for this could be the Topology and Orchestration Specification for Cloud Applications (TOSCA), which is a formalized language for the description of cloud and network services published

by the industry group Organization for the Advancement of Structured Information Standards (OASIS).

The actual MANO framework is made up by three seperate components. These are Virtualized Infrastructure Management (VIM) units, possibly several VNF managers and, naturally, an orchestrator module.

VIMs provide an interface to the virtualized infrastructure and allow the allocation of any resources needed for the deployment of VNFs. Such virtual infrastructure could be somthing like computing or storage facilities as well as virtual network links to enable the communication between deployed VNFs. Where and how the requested resources are to be deployed physically is the domain of the NFV Infrastructure (NFVI) unit. This is typically another framework independent from MANO and merely provides the virtualized abstraction of physical hardware that is necessary for NFV. If the orchestrator should be in charge of deciding where to deploy a VNF physically, it has to provide the necessary interfaces for retrieving information about the virtualized infrastructure and its current state.

The VNF managers are responsible for supervising Fault, Configuration, Accounting, Performance and Security (FCAPS) for VNFs via the associated Element Management System (EMS). Therefore, also lifecycle-management and VNF monitoring falls into their responsibility.

As the core component of MANO, the orchestrator unit itself determines the functionality of the complete NFV platform. He is in charge to decide how many and what type of VNFs should be deployed and should provide all other components with the necessary commands they need to fulfil the task in their respective domain.

To consolidate the understanding of the ETSI reference architecture and its implications on this work, the following section provide an overview of state of the art implementations and their limitations.

2.2.2 Virtualized Infrastructure Management (VIM)

Even though Kubernetes is known to be an orchestration platform for containers, it is not eligible as a VIM for NFV as it operates on application layer and has no built-in support for SDN or other network related tasks. Its main field of applicaton is the deployment and orchestration of Docker containers in centralized data center for cloud computing. The automatic scaling of deployed containers based on metrics

such as CPU utilization or request rate is prossible and the necessary load balancing is provided by the platform without the need for user interaction.

Kubernetes is operating on application layer and the monitored and load balanced request are typically full scale Hypertext Transfer Protocol (HTTP) connections. This makes Kubernetes in its entirety not a great fit for acting as VIM but one of its core compents may be able to serve an important purpose. That is the built-in solution for distributed storage *etcd*.

The most commonly used solution for Virtualized Infrastructure Management (VIM) is OpenStack, a framwork for cloud computing providing an on-premise solution for Infrastructure as a Service (IaaS). This framework is deployed in a client-server relationship and provides various modules that provide the allocation of storage or the execution of task on configured clients and can also create virtualized connections between nodes via SDN.

OpenStack as a framework is subdivided into several (optional) modules. Neutron is such a module and provides the necessary functionality for creating virtual links between nodes by deploying OvS instances and serving as a SDN controller where needed. Nova on the other hand allows the deployment of virtual machines and containers with supporting various hypervisors.

Together, Neutron and Nova provide the necessary functionality for serving as VIM and introduce an abstracted view of the underlay network. But just like Kubernetes and other cloud computing solutions offered by IaaS vendors like Amazon and VMware, OpenStack in its standard configuraton is providing the abstracted infrastructure layer without any specific awareness of the underlay network and other physical hardware.

Even though underlay network awareness could be introduced by incorporating other SDN solutions like ONOS as introduced in section 2.1.4, the network monitoring and latency away placement of VNFs remains on open problem in state of the art.

2.2.3 Management and Orchestration (MANO)

With Open Source MANO (OSM) and OpenBaton, two state of the art projects are available that implement a MANO framework and together with VIMs like OpenStack provide a complete NFV platform as introduced in section 2.2.1.

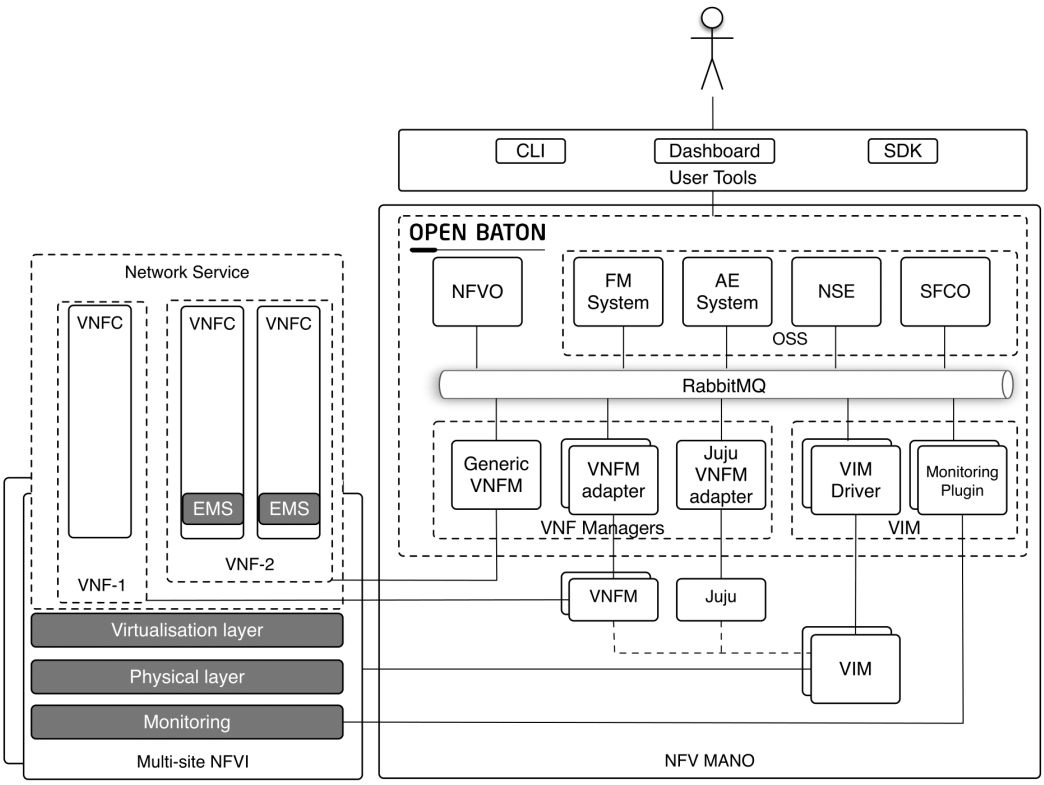

Figure 2.6: Architectorial overview of OpenBaton (Image source: [5])

OpenBaton is developed as a reasearch project on MANO in a joint cooperation of Fraunhofer FOKUS and TU Berlin. Its architecture is shown in figure 2.6 and features some similarities with the ETSI reference as all standard components NFV Orchestrator (NFVO), several VNF Managers (VNFMs) and the VIM units are represented.

We see in further detail, how VNFMs interact with the Element Management Systems (EMS) deployed inside those VNFs using the generalized VNFM unit provided by OpenBaton. Those EMS units implement the interface specification used by the generalized VNFM for life-cycle management and monitoring. VNFs using specialized VNFMs like Juju (another container orchestration system provided by Canonical) do not need EMS units as VNF and VNFM already follow the same interface conventions.

Another interesting aspect is the provision of a monitoring plugin that is missing in the ETSI reference. At the time of writing, the monitoring plugin is using Zabbix

(an enterprise network monitoring tool) that is providing the NFVO with statistics on CPU utilization and other metrics local to each VNF but can not collect any information on the physical underlay network of the VIM as this unit is supplied by OpenStack and it was already stated above that such a deployment is agnostic of the underlying hardware structure.

Open Source MANO (OSM) is the reference implementation of NFV by ETSI and currently available in its fifth release. Its most interesting feature is SDN assist that was introduced in [14] as a mechanism to provide SDN functionality for VIMs just like ONOS would serve as SDN framework for OpenStack via the ML2 interface as mentioned in a previous section.

With this feature, OSM would be fit for orchestrating latency aware VNFs as it could provide SDN for a common VIM solution and thereby implement the underlay network awareness itself. This could be possibly done with incorporating the monitoring solution proposed in this work.

2.3 Network Monitoring Solutions

As end-to-end latency is the Key Performance Indicator (KPI) of this work and current NFV platforms offer no functionality to collect the relevant information, it was necessary to investigate the current state of research on latency measurement in SDN environments, where using traditional utilities like sending Internet Control Message Protocol (ICMP) messages with `ping` is not possible or would result in a huge control plane overhead.

The earliest relevant publication on this topic is “Monitoring latency with OpenFlow” [15] as of 2013, wherein the first attempt on measuring the latency between SDN switches using OpenFlow messages was described.

The authors of [15] create a timestamped Ethernet packet in the controller and send it over the desired link using a `PACKET_OUT` message. To ensure that the packet will be forwarded to the controller again via `PACKET_IN`, an arbitrary Ethernet-Type is used on packet creation. After receiving the probe packet back on the controller, the time difference is calculated and the respective controller-to-switch latencies are subtracted. These values were in the meantime measured as the delay between `STATISTICS_REQUEST` and `STATISTICS_REPLY` sent to the switches. The result is the one-way latency of the link under test.

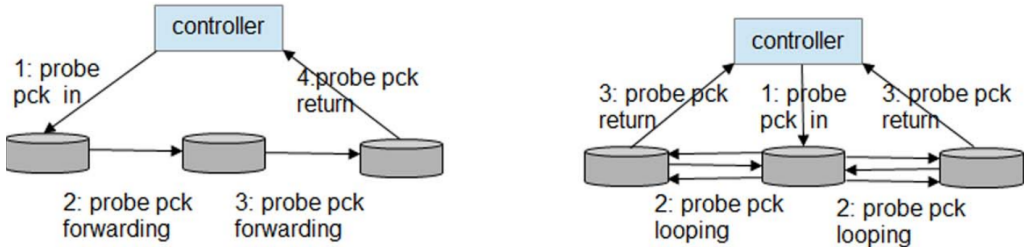

Figure 2.7: Conventional probing (left) and LLDP Looping (right) (Image source: [6])

Based on this idea, Liao et al. propose in "LLDP Based Link Latency Monitoring in Software Defined Networks" [16] the use of timestamped LLDP instead of plain Ethernet packets to minimize the control plane overhead as some sort of topology detection is necessary in most real systems.

The same authors extend this idea in "An Efficient and Accurate Link Latency Monitoring Method for Low-Latency Software-Defined Networks" [6] with a Looping scheme depicted in figure 2.7 (right) to eliminate the controller-switch latencies from the actual measurement.

With this, a timestamped LLDP packet is looped between two nodes while using Time-to-Live (TTL) for keeping track of the progress. According to the authors, this approach has the further advantage of improved scalability. This is due to the fact that in the convenientional probing scheme shown in figure 2.7 (left), a congested control network would decrease the measurment precision through increased delays inside the SDN controller. Considering network scale, congestion in the control network becomes more likely when more switches are attached to the given controller. With using LLDP Looping, the measurement of a link latency becomes independent of the amount of traffic in the control network.

2.4 The Serverless Computing Paradigm

With the so called *serverless* paradigm, a new concept for managing virtual machines and containers in cloud computing environments was introduced in recent years. This term by no means refers to the absence of servers in cloud computing, but rather to the automatic deployment of virtualized infrastructure (including servers) based on the incoming usage requests. In that sense, the server that is actually hosting some deployed functionality is effectively hidden from the admin-

istrator (or developer) and the adminstration responsibility is transferred to the cloud platform itself.

The general concept of state of the art serverless platforms presented in [17] and [18] shows that the typical deployment unit of a serverless platform is s *function*. This fact has introduced the term Function as a Service (FaaS) as a description of the service that serverless platforms provide. In contrast to IaaS, where virtualized infrastructure is provisioned *as a service* under the clients administration, FaaS guarantees the given function to be available on incoming requests without the need for further interaction with the client.

The invocation of a function is implemented just like the usage of library functions in regular shared libraries. The call signature should be well defined and is usually derived from the bare source code of the function that the platform expects for deployment. Such a function is invoked on an incoming HTTP request, which means that serverless platforms operate on application layer.

The internal performance metrics of a serverless platform are described in [17]. As the serverless paradigm allows for a more precise billing of customers (based on the number of requests or function runtimes) and at the same time tries to keep its available resource utilization as low as possible, a very important performance metric is the time period a deployed function remains in its active state before being destroyed due to exceeding an idle timeout.

An earlier destruction has the advantage to keep the resource utilization low and minimize the cost for the clients but may incur the disadvantage of a higher startup time for new requests as they can not be routed to an active instance of a function container. Function containers that are kept alive without immediately processing requests are also referred to as *warm containers*.

As users of state of the art serverless platforms are often complaining about increades cost in comparison to regular cloud deployments, the optimal break-even point of this tradeoff seems to be still unknown. In general, and as publications like [19] confirm, a serverless deployment is most effective when the rate of incomng request show a high volatility.

Recently, the idea of incorporating the serverless paradigm into NFV platforms has been articulated by publications as [19] and members of the Computing in the Network (COIN) research group have expressed their interest in using the serverless paradigm for managing VNFs in the informal draft [20].

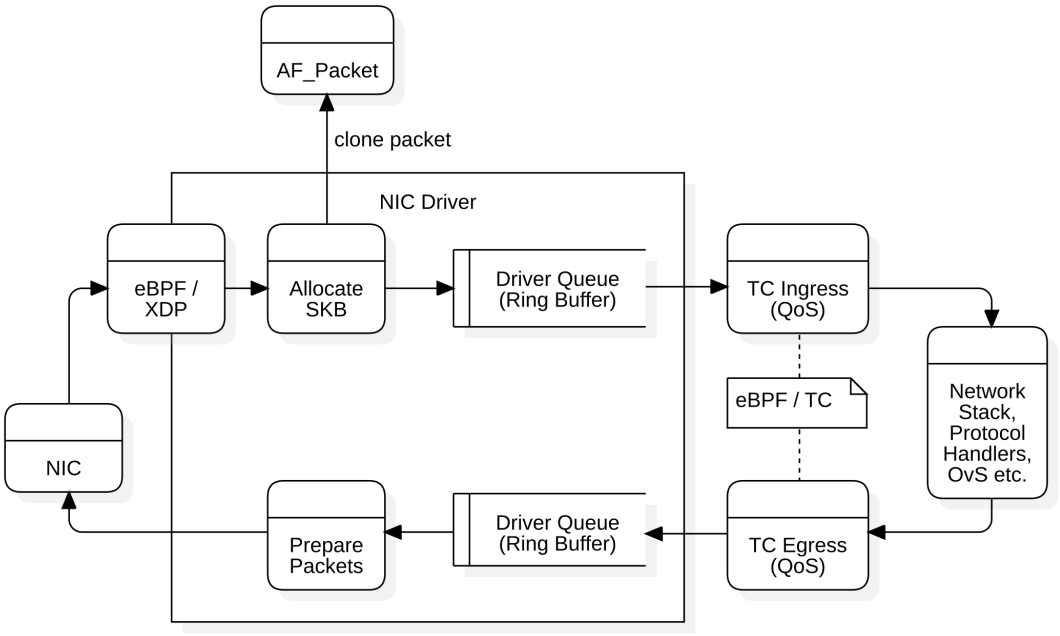

Figure 2.8: Data flow of network packets in the Linux Kernel

However, in what way such an incorporation can be realized is still an open problem and researchers face the difficulty of having to implore a radical change of architecture and concept on the very complex systems described above. To support further research on combingin NFV with the serverless paradigm, this work contains a proposal in section 4.2 on how to incorporate the latter in OpenFlow based NFV platforms.

2.5 The Extended Berkleey Packet Filter (eBPF)

Another important trend that may be used to improve the scalability and adaptability of NFV platforms is the Extended Berkleey Packet Filter (eBPF). That is an in-kernel virtual machine used for executing small programs inside a running kernel.

The eBPF somewhat resembles the Java Virtual Machine (JVM) in its way of defining a hardware independent instruction set that represents a target architecture for high-level programming language compilers. eBPF programs are usually written in the C programming language and can be compiled for the eBPF virtual machine

with widely used toolchains like GCC or LLVM. The resulting binary objects can be attached to hooks provided by the kernel whereas each hook is registered for certain type of event that triggers the invocation of the attached program.

Certainly the most important hook for attaching eBPF programs is defined by the eXpress Datapath (XDP). Figure 2.8 shows the location of XDP in the Linux kernels network packet data flow. If the Network Interface Card (NIC) driver supports the XDP hook, an eBPF program attached to XDP is invoked as early as prossible after a network packet arrived at the NIC.

In fact, the invocation of XDP happens even earlier than the allocation and enqueueing of the buffer structure (SKB) used for packet management inside the Linux kernel. The raw incoming data (in network byte order) can be read and manipulated by XDP as it arrived at the NIC.

After the processing is done, the XDP has to return a verdict over the packets further lifetime in the kernel. This is an important difference to regular raw sockets (`AF_PACKET` in figure 2.8), for which packets are duplicated and which cannot interfere with further processing of packets in the network stack. The possible verdicts are listen below. On error, the program may return `XDP_ABORTED`.

- Let the packet pass the XDP and further traverse the data flow depicted in figure 2.8 (`XDP_PASS`)
- Immediately return the packet to its ingress interface (`XDP_TX`)
- Redirect the packet to one or more other interfaces (`XDP_REDIRECT`)
- Drop the packet (`XDP_DROP`)

These operations allow XDP to implement network services with minimal overhead and maximized throughput. In case of simply dropping packets without further processing, "XDP offers a five-fold improvement over the fastest processing mode of the regular networking stack" [21]

For the increased speed and universal deployment (as the eBPF architecutre is hardware independent), XDP has been embraced by large companies like Facebook and Cloudflare for the implementation of network services like Distributed Denial of Service (DDoS) prevention and firewalls.

Before being attached to any hook, a eBPF program has to pass a validation test carried out by the Linux kernel. This test should ensure the stability of the kernel and restrict eBPF programs to their reserved adress space. To make validation

easier, certain limitations are implored on eBPF programs. In that regard, it is not allowed to call functions, loops must be unrolled or are limited to a certain amount of iterations and pointer check must be carried out by the developer. To provide a certain amount of functionality for XDP programs, the Linux kernel contains a library of so called *helper functions* that make a certain subset of kernel functions available to the eBPF program depending of the type hook it should be attached. It is not possible to call arbitrary kernel functions beside the especially encapsulated helpers already mentioned.

eBPF programs are invoked on events related to the registered hooks. As such, it is not possible to maintain a program state solely inside a program. For that purpose, it is possible to define *maps* of various types that reside in kernel space but can be read and modified from user space. Such maps allow to store state between program invokations and to communicate with user space utilities. Without further efforts, only the user space program originally defining a map has permanent access for reading and writing. When that user space utility exits, the reference to the map is lost and no other program can gain access any more. However, loosing the reference in user space does not affect the connection between the eBPF program and an associated map. To make such maps generally available for more than one user space program, they can be pinned to the `/sys` filesystem provided by the kernel.

In case the driver does not support the XDP hook or the manipulation on a “higher level” is desired, eBPF programs can also be attached to the Traffic Control (TC) hook provided by the kernel. Unlike XDP that only handles incoming packets, the TC hook is available for in- and egress. It is also possible to have both a XDP and TC program attached to the same network interface at the same time.

This works will use both hooks (XDP and TCP egress) to implement the comprehense network monitoring system described in section 4.3).

3 Task Specification

3.1 Research Question

As described in chapter 2, state of the art industry solutions and academic research are somewhat drifting apart nowadays.

One one hand, many industry groups are working on concrete implementations of NFV platforms, but they face the need to carefully plan for the requirements their customers might impose on the developed solutions. Such requirements might concern the multi-tenancy, security, standardization and ease of deployment as well as the support for a rich set of hardware platforms. All these requirements emerge from business considerations and customer care and are not directly related to academic research on the field of NFV.

Nonetheless, they impose huge difficulties for the devopers of industry ready NFV platforms. Consequently, such projects have become quite complicated constructs over time and even though some of them are openly sharing their source code and an interested researcher might theoretically have the opportunity to use this code to carry out research on NFV, the consideration of the aforementioned business requirements make current state of the art solutions not very usable for active research as they require too much time the get familiar with and and are typically not easy to modify.

This is basically the same problem, developers face themselves when using some third party frameworks. It is common knowledge in the industry that frameworks can make the task they were intended for very easy to solve but impose great difficulties on the developers when they try to leave the predefined path of develoment.

On the other hand, academic research is looking at NFV in more general way and does not have to consider business related restrictions, but is currently divided in two groups which are approaching NFV from different perspectives.

The first group is mainly concerned in motivational topics and is activley searching

for feasible use cases and future directions. They also try to abstract NFV platforms from concrete implementations and business considerations to develop a general idea of the conpect to identfy future trends and applications. Furthermore, they also take part on the process of standardization of interfaces and platforms.

The second group of researchers is mainly concerned with the mathematical problem of optimizing the placement and scheduling of functions on a set of interconnected workers. Finding suitable optimization strategies for NFV (and other related fields) is a highly mathematical challenge as the problem of function placement and scheduling has been proven to be NP-hard, that is a problem considered to be not possible to solve in deterministic polynomial time (with the emphasis on not deterministic).

For both groups, it is desirable to develop an understanding of what the restrictions of a real world implementation of NFV (but not the associated business considerations mentioned above) might impose on their research. Naturally, this is due to the inceasing complexity of state of the art solutions as NFV researchers should not be occupied with dissecting unrelated business logic. Consequently, this circumstance makes it very difficult to validate and, most importantly, compare different approches in current research. Especially in the field of placement and scheduling, the individual solution strategies are oftentimes basing on some assumptions and then proven by simulations that honor the same assumptions as well which leads to lacking comparability between active lines of research.

It should therefore be the ambition of this work to shed some light on the gap between state of the art solutions and recent proposals in current research. This is should be done by designing and building a lightweight network softwarization platform that is able to optimize for end-to-end user latency as its KPI by incorporating emerging trends of other fields of research.

How to create a platform that is able to optimize for latency is a research topic in its early days. To the best of the authors knowledge, there has no solution for this problem been published previous to this work.

3.2 Design Goals

Conventional NFV platforms are derived from cloud computing frameworks that were originally designed to provide a high level of abstraction. In consequence,

state of the art solutions are approaching NFV in a top-down fashion that made it difficult to implement the awareness of the underlay network that is needed for use cases like a latency-optimized deployment of VNFs in the network.

By build a network softwarization platform from ground up, the author is approaching NFV from the opposite direction as the state of the art. This should allow to investigate fundamental challenges in NFV without the business related overhead of industry-ready platforms and provides researchers with a lightweight and reproducible solution for evaluating new ideas and paradigms.

Additionally, the system developed in this work should serve as a Proof of Concept (PoC) for the authors effort to overcome two major issues of conventional platforms.

The first issue to be addressed is the missing awareness of the underlay network in the Virtualized Infrastructure Management (VIM) solutions currently available. This is proposed to be overcome by the monitoring subsystem proposed in section 4.3 that is providing a comprehensive monitoring of parameters like link latency and CPU utilization by incorporating the eXpress Datapath (XDP) provided by the Extended Berkleey Packet Filter (eBPF) that was recently developed by the Linux community.

The second issue to be addressed is the lack of agility in state of the art platforms as they only allow predefined deployments of VNFs and even though such deployments have recently become scalable in the sense that VNFs under high load are automatically duplicated and load balanced, the complexity of administrating such deployments is still very high. This issue is addressed in section 4.2 by proposing a MANO framework following the serverless paradigm presented in section 2.4 that allows an automatic deployment of Service Function Chains (SFCs) as the need arises and thereby transferring the responsibility for administration from clients to the platforms itself.

To summarize the aforementioned aspects, the system developed in this work should be aligned to the following design goals.

- To enable future use cases that depend on a low end-to-end service latency, the platform should support the latency aware embedding of SFCs in the underlay network
- To minimize resource usage and administration complexity, placement decisions, traffic engineering and VNF instantiation should be carried out auto-

matically as the need arises

- To ensure extensibility and scalability, the developed platform should minimize dependencies and coupling between modules and exhibit a clear formulation of responsibilities
- To support neatless integration of future use cases, no assumptions concerning the behaviour of connected clients should be made and no dependencies should be imposed on them

To the best of the authors knowledge, this is the first attempt to build a network softwarization platform following the aforementioned design goals.

4 Design of a Serverless Network-Softwarization Platform

4.1 System Overview

Before presenting the individual subsystems in details, this sections provides an overview of the system design and its relation to the ETSI reference architecture.

The component diagram in figure 4.1 depicts the modules of which each subsystem is made up by and their respective dependencies of each other.

The core of the platform is represented by the MANO subsystem to be seen in the second column. The components NFVO, NFV Manager (NFVM) and VIM are directly related to the ETSI reference previously shown in figure 2.5. In addition to this, the responsibilities and interconnection of these components in the architecture proposed by this work enable the serverless management of VNFs that is supposed to improve resource usage and reduce administration complexity as formulated as a goal in the previous section.

The NFV Orchestrator (NFVO) is the main building block of the platform which coordinates all other subsystems. He is implementing a SDN controller for the virtualized network connecting compute nodes and clients. Accordingly, all traffic engineering decisions and the necessary configuration of SDN switches falls in the responsibility of the NFVO. For monitoring purposes, he is in charge of initiating probe requests and using the collected results for maintaining a network model. This information is used for placement decisions arising in SFC handling that is also a responsibility of the NFVO.

The NFVO has a complete picture of the virtualized network and the available compute nodes but has absolutely no information on how incoming traffic is handled inside the compute nodes. This is the responsibility of the NFV Manager (NFVM) that stands as a deputy for the NFVO and forwards incoming traffic to the appropriate VNF while maintaining the correct Network Service Headers (NSHs) for each

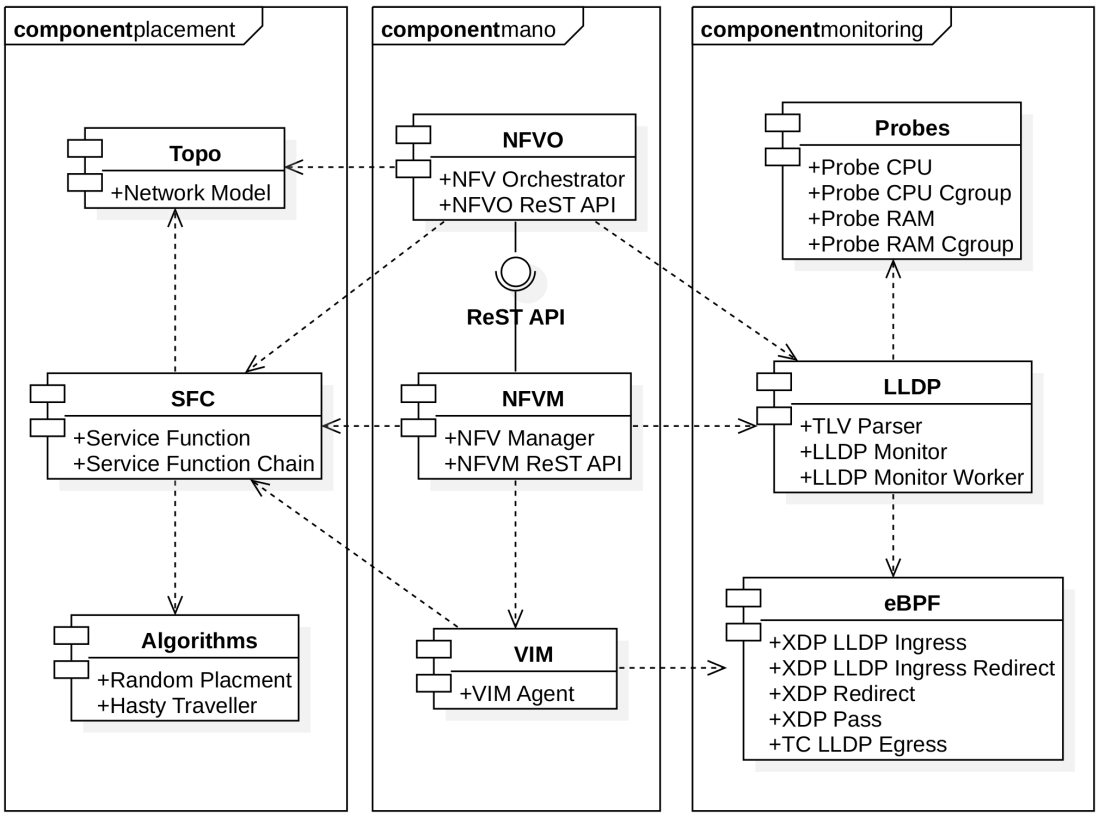

Figure 4.1: Major components of each proposed subsystem

deployed VNF. The NFVM also handles VNF deployment requests via a provided Application Programming Interface (API) following the paradigm of Representational State Transfer (ReST).

The deployment of VNFs and the provision of necessary infrastructure inside a compute node is delegated to the Virtualized Infrastructure Management (VIM).

In the ETSI reference, VIM is supposed to provide an abstract interface to the virtualized infrastructure including the SDN switches that connect the individual compute nodes to each other. In contrast to this, the VIM proposed in this work is implemented as an agent running on each compute node individually being only responsible for the local management of VNFs and the necessary intra-node infrastructure. The task of managing SDN switches and traffic engineering is hereby transferred to the NFVO. In that sense, the NFVO becomes some sort of a high level VIM himself, enabling him to combine monitoring, placement and traffic engineer-

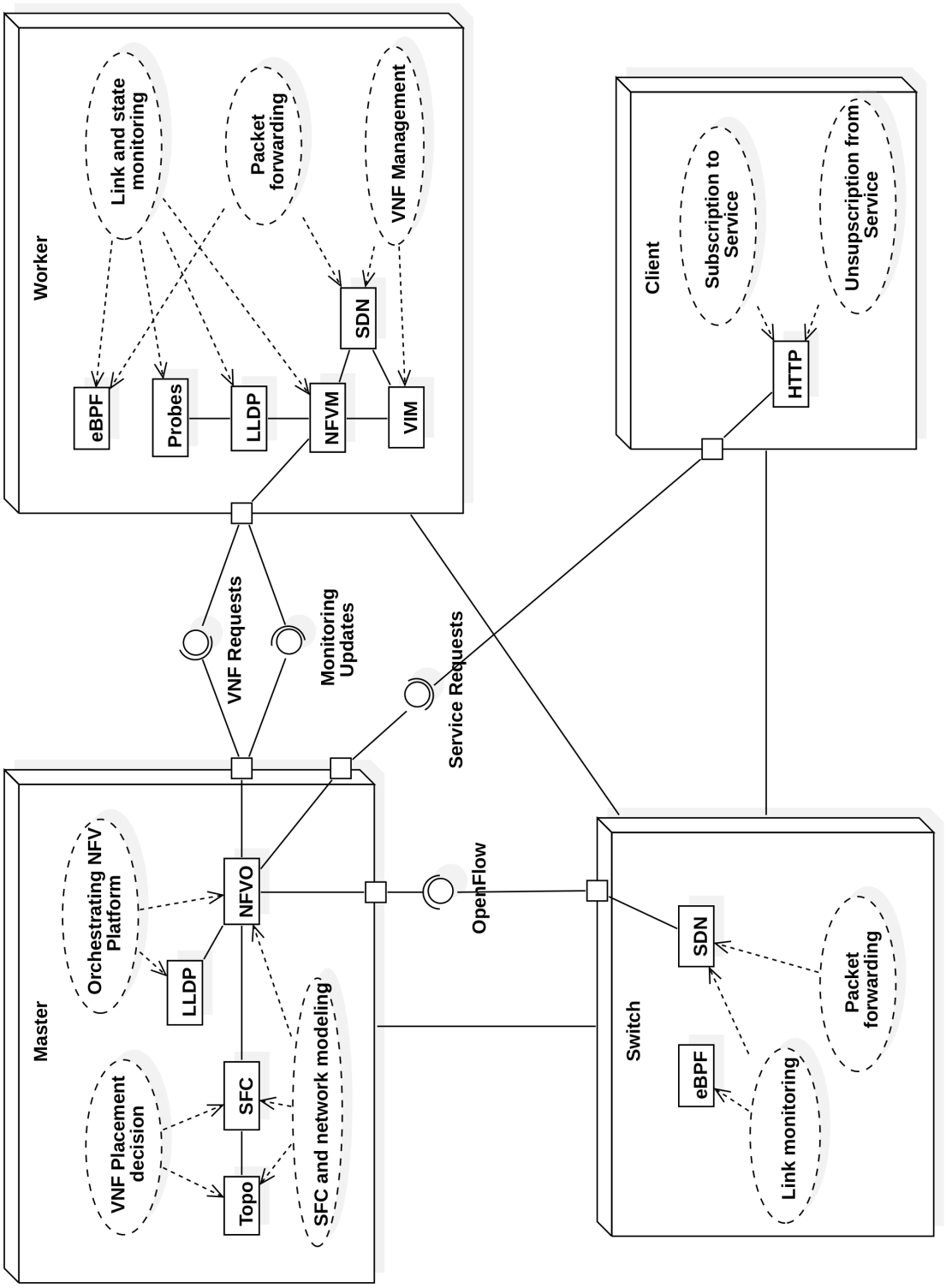

Figure 4.2: Deployment model of the proposed NFV platform

ing to provide automatic and topology aware SFC embedding.

The components contained in the MANO subsystem depend on other modules from the monitoring and placement subsystems as depicted in figure 4.1. The modularization choosed in these components is not related to a general architecture of network softwarization, but merely the result of designing a PoC implementation with minimal overhead. It should be clear that the tasks these modules solve must be solved some way or other, but there is certainly room for individual design choices. However, the concept of those submodules (and especially the monitoring solution that has no equivalent in the start of the art) applys to the general architecture proposed in this work and is discussed in the following sections.

But before that, it is necessary to take a closer look on the envisioned deployment of the proposed platform in the network, as this is a very important aspect for understanding the general architecture.

The deployment diagram in figure 4.2 shows the multiple roles, in which the individual modules of the platform have to perform. These are *Master*, *Worker* and *Switch*, wherby multiple workers are connected via switches to a master that controls the platform. Figure 4.2 also contains a *Client* node, but as one of the goals formulated above was that no dependencies should be imposed on clients, just as simple ReST API for service subscription is provided by the master that can be used by any HTTP client available. The traffic between clients could be following any protocol desired. Deployed network services are then intercepting such traffic without any feedback to these clients and require no further interaction.

In the PoC implementation, switches are controlled by masters via the OpenFlow protocol as they are represented by (softwarized) SDN switches. This is, strictly speaking, an implementation detail and other protocols, if existing, could also be used for the purpose of SDN configuration.

Just the same relation as between clients and masters, where a ReST API is provided for service registration, also workers and masters are communicating requests for VNF deployment through a ReST API provided by each worker. This concept ensures decoupling masters and workers as each has a strict set of responsibilities and tasks are dispatched via a well defined stateless interface. In that sense, every task related to complete SFCs and the global view of the network lies in the masters responsibility and each task concerning individual VNFs (of which SFC are made up by) are dispatched to the respective worker responsible for deployment.

4.2 Management and Orchestration Subsystem

4.2.1 Network Services

Before having a detailed look how the MANO components interact to provide serverless NFV functionality, it is necessary to define what a network service actually is and how it should be handled.

For this work, a network service is defined as a set of operations that are applied to network packets as they move across the network from a source to a destination client. From platform perspective, source and destination are both *clients* of the platform but of course could assume any role in relation to each other, for example *server* and *client* in case of a video stream that is transported through the network and an operation like transcoding is applied to the stream as a network service.

As a requirement resulting from the design goal of not imposing any dependencies on the connected clients, network services should be provided in such a way, that neither of the clients between the service is registered should get notice of its presence. Therefore, this design of network services somewhat resembles the idea of Man-in-the-Middle attacks and may thus arise security concern that has to be addresses by future research.

For modeling and implementation, a nework service is represented by a Service Function Chain (SFC) consiting of one or more Virtual Network Functions (VNFs) (or service functions) that represent the individual operations the network service should comprise.

As previously mentioned, network services are in this work defined between two clients of the platform. In consequence, the classification of packets belonging to a deployed SFC is done via source and destination IP address. This design choice was made for the sake of simplicity and in future implementations of the proposed concept, it is perfectly possible to add more attributes to the classification process.

The implementation of service registration consists of a ReST API provided by the NFVO, where service requests can be sent to in form of a service description complyig with the JavaScript Object Notation (JSON) standard.

Such a service description must provide the source and destination IP address, a list of operations to be applied to classified packets and may additionally specify some optional parameters like the placement strategy to be used or others. Section 4.2.4

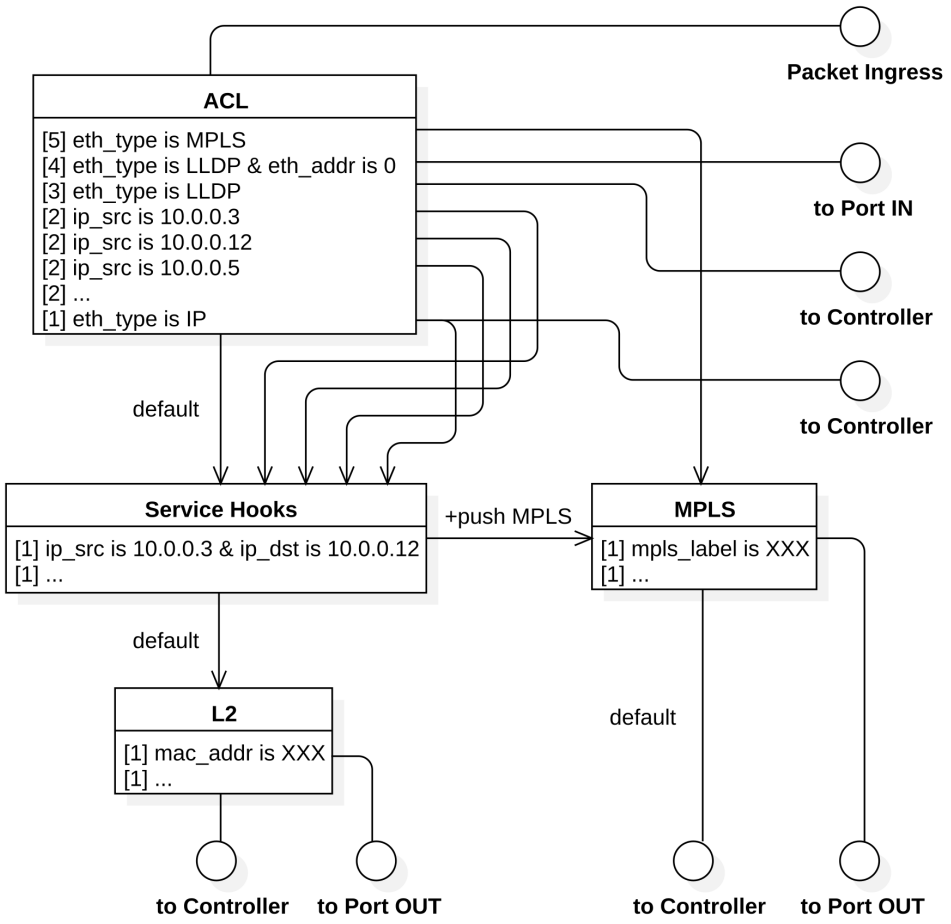

Figure 4.3: The data flow inside SDN switches was optimized to support detailed monitoring and serverless SFC embedding

provides more information on the design of operations and VNFs.

4.2.2 Traffic Engineering for NFV

The classification of traffic belonging to a deployed network service and the provision of optimized routes along the VNFs of whch the service is composed of defines the problem field of traffic engineering.

The interconnection of VNFs is often realized via IP tunneling. Recent efforts went into the direction of standardizing a Network Service Header (NSH) holding the necessary information to route packets belonging to a network sevice along predefined

paths in the network [22].

However, the PoC implementation of this work is based on the OpenFlow protocol that does neither allow the use of NSHs nor provides the necessary functionality for IP tunneling on the switches. Therefore, a search for alternatives brought the use of Multiprotocol Label Switching (MPLS) in consideration.

The MPLS protocol is supported by OpenFlow and allows to push arbitrary labels onto a packets header stack. This is usually done between the Ethernet and IP header. When a MPLS label is present, the Ethernet type is changed to a value specified the MPLS protocol. This enables fast identification of MPLS packets.

For the actual traffic engineering for NFV, this work proposes a SDN configuration as depicted in figure 4.3. This approach comprises the definition of four different flow tables with different responsibilities. The flow table entries in figure 4.3 only comprise the matching condition and are accompanied by their priority value. Highest priority rules are processed first.

The Access Control List (ACL) table is used for rules that have to be directly applied on ingress. Packets having already been tagged with an MPLS label are directly forwarded to the MPLS table, where all rules for the actual traffic engineering of deployed network services are set by the NFVO. These MPLS rules are of course individual to each switch in the network as the describe the path a packet has to take to visit each VNF along the way to its destination.

For the placement decisions described in section 4.4, it is necessary to know the switch on which each involved client is connected to (as these define the start and end point of the route through the network). This is possible through the application of the default action of the ACL table shown in figure 4.3. When an IP packet arrives at the switch and its source address is unknwon, it should hit the default action in the ACL table whereafter the packets is both forwarded to the next table and to the SDN controller, who hereby gets notice of a client with the given IP address connected to the given switch. When such information arrives at the controller, he subsequently installs rules in the ACL table that only forwards packets to the next table without giving the controller further notice about this.

The next table mentioned before is the *Service Hook* table. As its name suggest, ths table is used to store service hooks and results from the design choices described in section 4.2.3 for implementing a serverless VNF lifecycle.

If no service hook is matching the incoming packet, it is forwarded to a regular

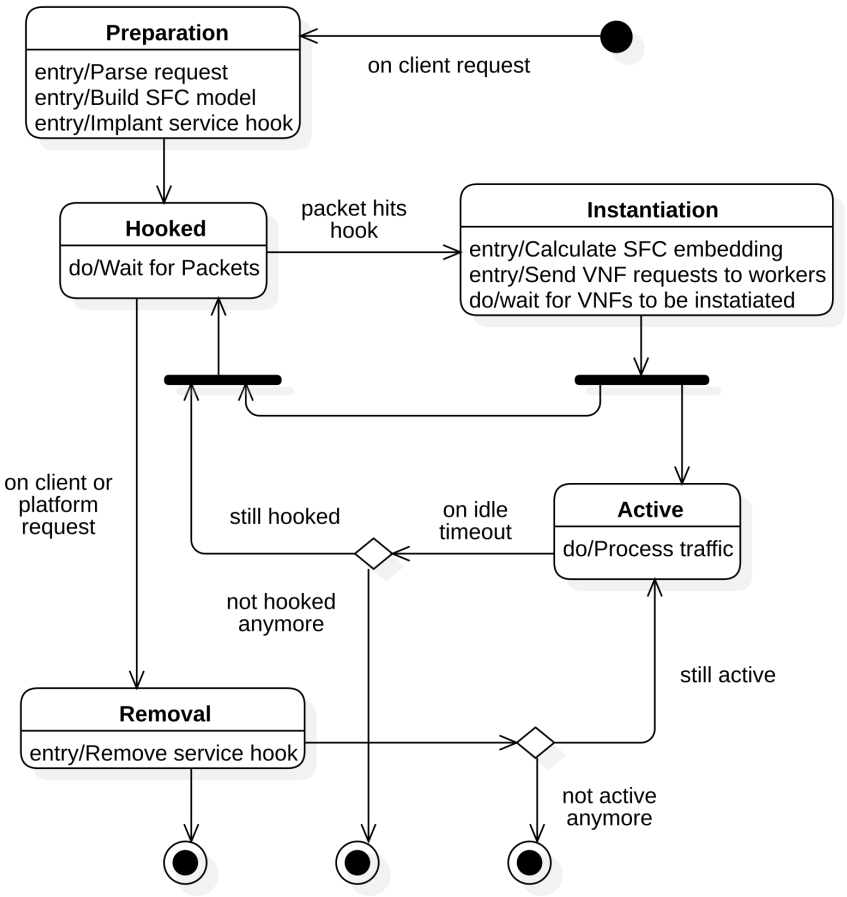

Figure 4.4: State diagram representing the serverless SFC lifecycle

Media Access Control (MAC) routing table to ensure normal communication in the network for packets not belonging to any network service or monitoring solution.

4.2.3 Service Lifecycle Management

The aforementioned service hooks are a concept developed by the author for supporting an automatic deployment and instantiation of SFCs following the serverless paradigm introducted in section 2.4. To that end, a similar approach as already implemented in SDN was chosen for the provision of requested VNFs

Service hooks installed on the switches representing the ingress point of a network service allow the notification of the NFVO module on incoming traffic. He can thereupon instantiate (and later remove) the requested SFC as the need arises. In that

regard, the provision of VNFs strongly resembles the handling of flow table rules for unknown packets as introduced by SDN in section 2.1. The developed concept for serverless SFC provision is summarized in the state diagram in figure 4.4 depicting the complete lifecycle of a network service.

Such a lifecycle begins with a clients network service request arriving at the NFVO. In case of the proof concept implementation built by the author, this is possible throuth a ReST API but other solutions could provide the necessary interface as well.

The service request is thereupon transformed into a SFC model[1] and a service hook is implanted on the switch representing the ingress point of the service. No other operation and no VNF deployment is carried out at this point of time.

A service hook is simply a flow table rule that matches the specified service attributes (like source and destination IP address in the simplest case) and is installed in the service hook table shown in figure 4.3. The actions defined on such rules are to push the NSH in form of a MPLS label onto the header stack of matching packets and subsequently forward it to the MPLS table.

After the service hook is installed, the service is considered to be *hooked*. When incoming packets match the installed service hook, they are forwarded to the MPLS table and once more forwarded to the NFVO if no matching NSH label was found in the MPLS table.

In that case, a packet belonging to an inactive network service must have arrived at the switch. The NFVO thereupon calculates the embedding strategy specified in the service description (latency-optimized, random placement, etc.) and sends deployment requests to the individual workers selected for hosting the VNFs. After all requested VNFs have been successfully deployed, the individual MPLS rules are installed along the calculated path and the original packet that hit the service hook is forwarded along the newly created path. Future packets hitting the service hook will find appropriate entries in the MPLS table and will therefore not be forwarded to the NFVO as long as the SFC is still active.

For an automatic removal of the deployed SFCs, all NSH routing entries in the MPLS table are accompanied with an idle timeout value. As standardized by OpenFlow, rules featuring an idle timeout will be automatically deleted after the specified period has past without any packets matching the rule.

[1] The SFC model is described in section 4.4

In contrast to the automatic removal of MPLS switching rules, the service hooks of registered network services will only be removed on user or platform request. In order not to loose any packets that are currently travelling along SFC at the moment of service hook removal, only the hook itself, but not the switching rules installed during the SFC instantiation will be removed. Without a service hook, no packets can enter the SFC any more and after the idle timeout sets in, the deployed SFC will be completely deleted automatically.

Service removal request can not only be issued by clients, but also by the platform itself. This mechanism may used when the platform detects significant changes in the network model that make the placement decisions calculated before obsolete. This may occur when links are flooded or workers are overloaded. In such a case, the platform may issue an alternative deployment of a given SFC under a new NSH label and subsequently revoke the service hook of the previous SFC deployment. Packets hitting the service hook will then be routed along the new SFC deployment and packets that were still travelling along the previous route may finish their processing without disturbance. As described above, the previous deployment will be automatically removed after the idle timeout sets in.

In that regard, the differentiation of the state of a SFC into *hooked* and *active* as shown in figure 4.4 together with the optimized data flow inside the SDN switches shown in 4.3 is the architectural key concept for enabling a serverless SFC deployment. The coupling between the components of the proposed NFV platform is minimized by delegating every task related to individual VNFs to the NFVMs residing at their place of deployment while the NFVO is solely responsible for managing complete SFCs and their lifecycle.

4.2.4 Service Embedding

After having described the SFC lifecycle management from the NFVOs perspective, this section provides a description of traffic engineering and VNF instantiation from the NFVMs and VIMs perspective on each respective compute node (or worker).

A real world deployment of the PoC implementation for a simple network with one master and two workers is depicted in figure 4.5.

The master node is simply a worker running the NFVO and providing the accompanied ReST API whereby the NFVO also acts as a SDN controller for the switches interconnecting clients and workers.

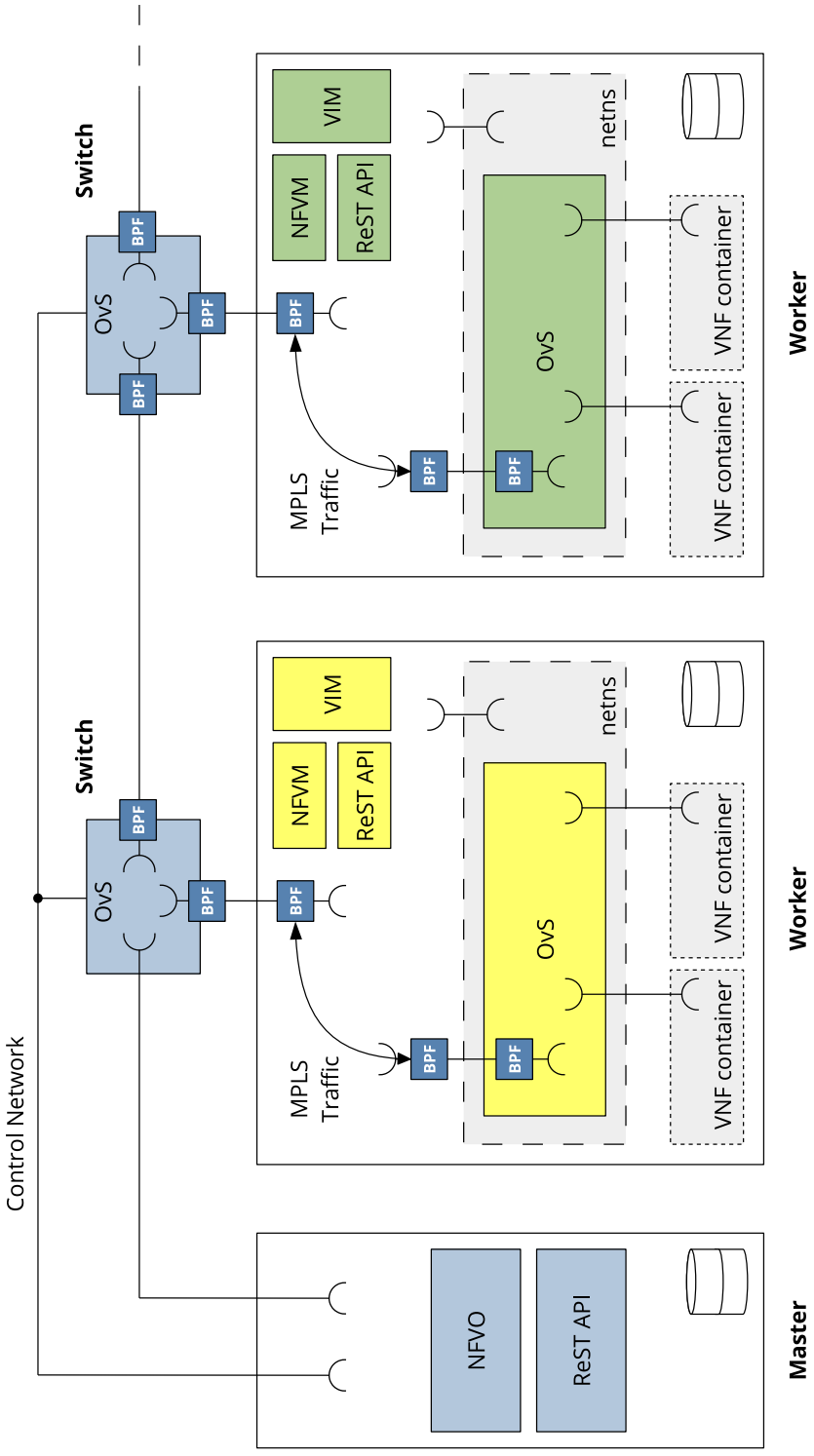

Figure 4.5: Example Deployment

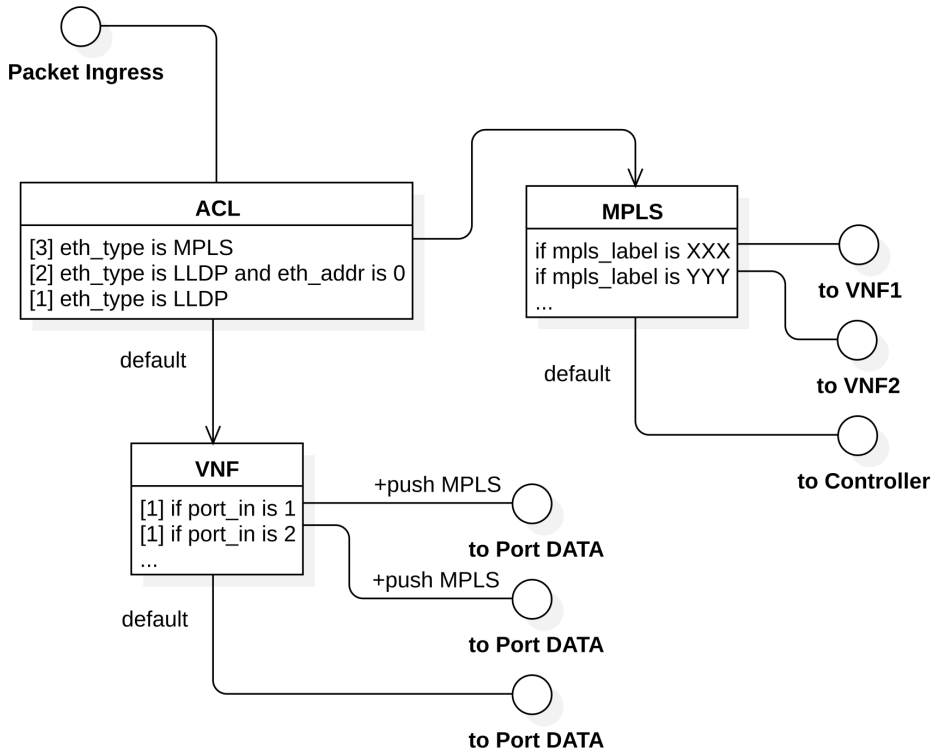

Figure 4.6: The data flow inside per-worker deployed virtual switches

Very similar to that, also the NFVM deployed on each worker is acting as a SDN controller, but not for the overlay network switches, but for the local virtual switch deployed inside a seperate network namespace as shown in figure 4.5.

Namespaces are a mechanism provided by the Linux kernel for restricting the view of certain resources for individual processes. In case of a network namespace, a process belonging to this will not have any access to the workers root network stack and interfaces, but will instead maintain its own set of interfaces that could have been provided by the administrator by having real network interfaces moved to the namespace or creating virtual ethernet pairs and moving one of the endpoints to the namespace.

In the PoC implementation, this feature is used for encapsulating the local virtual switch and its connections to the individual VNFs deployed on the worker. This encapsulation simplifies the seperation of network traffic belonging to deployed SFCs and other traffic that is directed at the worker himself, for example a VNF deployment request issued by the NFVO.

A XDP program is attached to the main network interface of each worker that is (besides its main functionality of link monitoring described in the next section) used for forwarding all packets featuring a NSH in form of a MPLS label to the data interface of the locally deployed virtual switch. It is a restriction of XDP, that all interfaces to whom packets are redirected to by XDP, must also have a XDP program attached. This is the reason, why also the data interface inside the locally deployed virtual switch features an eBPF tag in figure 4.5. In this case, the attached program is doing nothing but pushing packets to the kernels network stack without any restriction. On its way out of the virtual switch, all packets are forwarded to the workers main network interface by another XDP program.

Responsible for the provision of the internal infrastructure of a worker is the VIM module. This module creates namespaces, virtual links and switches as needed for the operation as a *worker* of the proposed NFV platform.

The VIM is also responsible for instantiating the requested VNFs. These are usually containers (Linux or Docker containers) in the PoC implementation but could also be represented by other technologies like unikernels or even individual XDP programs. For now the PoC assumes the deployed VNFs to already be available in some way. Future research could be directed on the topic of exchanging VNFs in a distributed manner.

The data flow of the workers virtual switches is very similar to that presented in figure 4.3, but are missing the MAC routing table as the switches are not used for arbitrary traffic. An overview is depicted in figure 4.6.

Incoming Packets featuring a MPLS label are stripped thereof and subsequently forwarded to the appropriate VNF. Packets returning from a VNF are again tagged with an approriate label and redirected to the MPLS table if the next VNF following in the respective SFC way also deployed on that worker. Otherwise, the packet would be redirected to the data interface and in consequence travels back to the overlay network.

The necessary rules are installed on the virtual switch by the NFVM. If he reveives a VNF deployment request, he prepares the VNF to be deployed but may only directly instantiate it when the request features an attribute to this effect. When a tagged packet is processed in the MPLS table inside the virtual switch and no matching entry is found (because the VNF was not yet instantiated), the packet is forwarded to the NFVM who immediately instantiates the requested VNF by incorporating the VIM module and installes appropriate en- and decapsulaton rules on the virtual

switch.

These rules are accompanied with an idle timeout value and the virtual switch notifies the NFVM when a rule was deleted due to timeout. In that case, the NFVM can have the VIM destroy the unneeded VNF.

4.3 Monitoring Subsystem

4.3.1 eBPF and LLDP Based Link Monitoring

The review of state of the art monitoring solutions in section 2.3 led to the conclusion that there is still no consesus among researchers on how to optimally monitor link latency in SDN networks. A latency-optimized network service embedding, as it is the goal of this work can only be carried out with a good system for measuring link latencies in the provided network.

To that end, this work proposes a comprehensive monitoring system based on LLDP Looping as first mentioned in [6] and previously described in section 2.3. But unlike the strategy proposed by the authors of [6], the solution of this work is able to measure asymmetric latencies as they occur on heavily congested networks and is able to cope with the various deployment contexts of the proposed NFV platform. This is enabled by the incorporating XDP as one of the emerging technologies envisioned by the author to improve the agility of future NFV platforms described in section 2.5.

In congested networks, the input queues of forwarding entities begin to fill up and forwarded packets incur higher sojourn times as they move along their route. As the two endpoints of a link may face different amount of traffic to be enqueued (for forwarding between multiple ports), this effect depends on the direction a packet is going across the link and leads to asymmetric latencies.

The actual link monitoring is carried out as depicted in the communication scheme in figure 4.7. The SDN controller (or NFVO in this work) sends out an especially crafted LLDP packet to each node in the network who will subsequently write a timestamp in one of the two Type-Length-Value (TLV) fields[2] prepared for carrying Round Trip Time (RTT) measurment results and forward the packet to its neighbor along the link to be measured. The neighboring node will forward the incoming

[2]The packet layout of a LLDP probe is shown in table 4.1

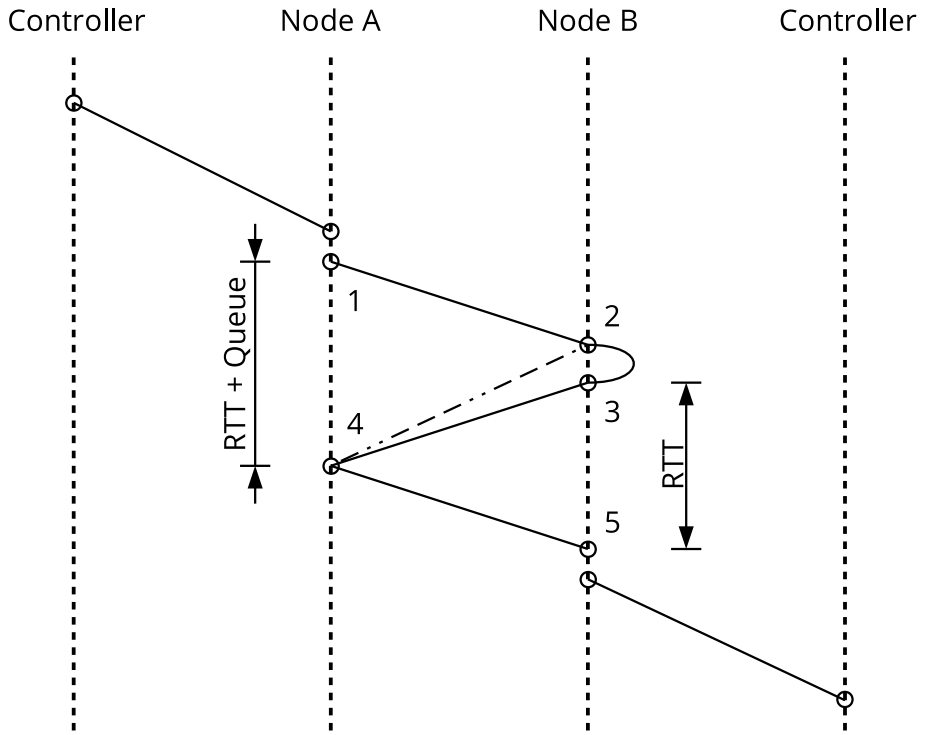

Figure 4.7: Communication scheme of LLDP Monitoring

packet up its network stack to the switching logic (i.E. an OpenFlow implementation). The switching logic was previously configured[3] to immediately resend the packets trough its ingress port. On its way out, a timestamp will be stored in another TLV field prepared in the packet. On receiving the packet again, the first node will calcualte the RTT in the first TLV field and resend it to its neighbor as fast as it can without further processing or enqueueing. In the same way, the neighboring node will calculate the RTT and store it in the respective TLV field, writes its MAC address into the source field of the MAC header and subsequently fordward the finished packet to its controller.

Before that last step, the source MAC address will be empty. This fact is used in the ACL tables for distinguishing LLDP packets to be looped from LLDP packets to be forwarded to the controller. This is necessary, as OpenFlow does not allow to match packets based on TTL but this value is used for keeping track of the progress in LLDP Looping.

With this scheme, two RTT measurments were carried out. One between between

[3] See figures 4.3 and 4.6 for the LLDP rules installed in the ACL table

points (1) and (4) and another one between points (3) and (5) in the figure 4.7. It is to be noted that the first RTT value includes the queueing delay incurred at the neighboring node, while the second node should only contain transmission and propagation delays.

According to [23, p. 46], the latency that packets incur on traversing a link between two endpoints can be modeled as the sum of propagation, transmission and queueing delay as expressed in equation 4.1.

$$Latency = T_{Propagation} + T_{Transmission} + T_{Queueing} \tag{4.1}$$

The propagation delay results from the limited speed of propagation on the given physical medium. In case of a transmission via electromagnetic waves, this delay could be calculated by dividing the speed of light by the physical link distance. As it is a physical constant, this delay is independent of the direction along the link.

$$T_{Propagation} = \frac{SpeedOfLight}{Distance} \tag{4.2}$$

The transmission delay is calculated by the division of the amount of data to be sent by the link throughput. Just like the propagation delay, this value is independent of link direction.

$$T_{Transmission} = \frac{Size}{Throughput} \tag{4.3}$$

But in case of the packets used in this work for link latency probing, this delay can be assumed negligible as a probe packet has a size of under 100 bytes and should therefore lead to transmission delays in the range of several microseconds or even less on typical links.

This insight led to the development of the communication scheme as shown in figure 4.7. The two combined RTT measurements allow the separation of queueing and propagation delay. With the following equations, the link latency in the forwarding direction can be calculated.

$$m_1 = RTT \tag{4.4}$$

$$m_2 = RTT + T_{Queue} \tag{4.5}$$

$$Latency = m_2 - \frac{m_1}{2} \tag{4.6}$$

Protocol	Field	Description
MAC	Source	MAC address of Node B
	Destination	01:80:c2:00:00:0e (LLDP earest Bridge)
	Ethertype	0x88cc
LLDP	Type	1 (Chassis ID)
	Length	9
	Value	DPID of switch (destination) if available
LLDP	Type	2 (Port ID)
	Length	17
	Value	UUID of compute node (destination) if available
LLDP	Type	3 (TTL)
	Length	2
	Value	3 -> 0 (used for identifying loop progress in eBPF programs)
LLDP	Type	127 (Organizationally specific)
	Length	12
	OUI	0x1ea5ed (specified by the author)
	Subtype	1 (specified by the author)
	Value	Measured RTT
LLDP	Type	127 (Organizationally specific)
	Length	12
	OUI	0x1ea5ed (specified by the author)
	Subtype	2 (specified by the author)
	Value	Measured RTT + queueing delay
LLDP	Type	0 (End of LLDPDU)
	Length	0
	Value	

Table 4.1: Structure of a LLDP probing packet

It is noteworthy that only the forwarding latency can be measured with one probing request. In a real world setup, the SDN controller (or NFVO) would send prope requests to all switches in the network. On receiving a request, a switch would flood all its ports with the probing packet and thus carry out the communication scheme described above with each of its neighbors. In this way, every link connecting two switches will be measured twice, but in different directions and thus the forwarding latency in both directions can be measured.

As the probing packets return to the controller from a different switch than they were sent to, this scheme allows combined link monitoring and topology detection.

The controller only has to send probe requests to all switches he knows of and can build a full network model from the probing results.

The separation of queueing and propagation delay is made possible by the incorporation of eBPF and XDP for implementing the necessary forwarding logic. State of the art prototypes of latency measuring solutions like [6] have modified the source code of user space applications like OvS for implementing LLDP Looping. In contrast to this, the author of this work decided to implementation extended version of this in form of XDP and eBPF programs. This approach promises higher throughput [21] and greater flexibility in the deployment context.

XDP programs attached to a network interface are immediately executed on packet ingress. This makes it possible to timestamp and return packets immediately without the enqueueing and processing normally done inside the kernels network stack and thereby preventing the occurence of queueing delays. This is used at point (4) in figure 4.7 for the measurement of an RTT value solely composed of the transmission and propagation delays. Without using XDP, this would only be possible with specially designed hardware components, which could arise difficulties when also links to compute nodes have to be monitored.

For a precise RTT measurement, it is important to be able to timestamp outgoing packets on the verge of leaving the network stack as necessary at points (1) and (3) in figure 4.7. This functionality can be achieved with an eBPF program attached to the TC hook provided by the Linux kernel.

Therefore, the communication scheme in figure 4.7 is realized in the PoC implementation in form of two eBPF programs attached to each interface connected to a link to be monitored. The first eBPF program is attached to the XDP hook of the given interface and handles incoming packets as early as possible (points (2), (4) and (5) in the diagram). The second eBPF program is attached to the TC hook of the given interface and handles outgoing packets as late as possible (points (1) and (3) in the diagram).

In addition to the new possibility to approximate asymmetric link latencies, XDP also allows to enable link monitoring on compute nodes (or workers) that would otherwise be hard to realize. The example of a real deployment in figure 4.5 shows the various deployment points of eBPF (or XDP) programs in the network. Attaching the workers locally deployed virtual switches directly to its main interface would arise the difficulty of having to communicate with the worker through an unconfigured switch (which is not possible) and to provide some way to connect these

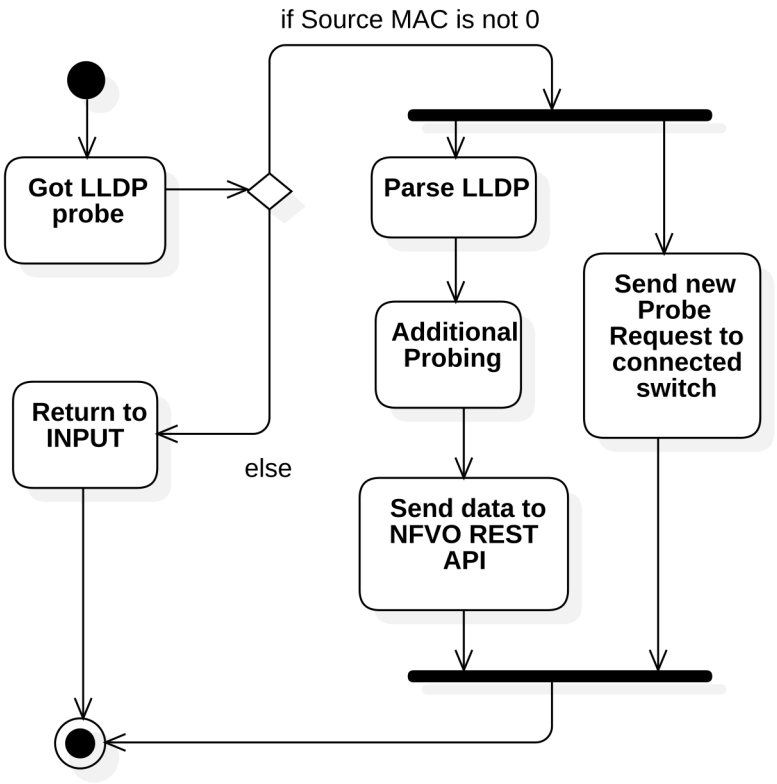

Figure 4.8: Activity diagram of a worker receiving a LLDP packet

switches to a SDN controller. With the deployment of a XDP program for redirecting only MPLS and LLDP traffic to the virtual switch (as only this traffic should belong to network services or probing requests), the seperation of management and service traffic is possible. Such a XDP program is implemented in the PoC as a simple extension of the program developed for the links between switches. This setup enables the VIM to easily bootstrap any compute node in the network desired for the deployment of VNFs.

Considering the LLDP probing packets, the NFVO has no direct access to the locally deployed switches and therefore can not send probing requests to these switches directly.

To that end, all LLDP traffic is forwarded by the aforementioned XDP program to the local switch who is configured to handle LLDP packets in the same way as the switches of the outer network (see figure 4.6). Thus, monitoring probes are forwarded to the NFVM instead of NFVO. The NFVM will immediately issue a probe

request to be sent the switch the worker is connected to and at the same time forward the received link information in form of a JSON serialized string to the ReST API provided by NFVO. Figure 4.8 provides an overview of this behaviour.

With this, also the links connecting computes nodes are measured in both directions and the NFVO is furthermore able to differentiate between switches and workers as monitoring results arrive at the NFVO via two different channels depending on the destination nodes deployment role.

4.3.2 Additional Compute Node Probing

In addition to the latency values measured via LLDP Looping, a compute node usually features various other performance metrics that are of interest while searching for an optimal SFC embedding in the NFVO. State of the art NFV platforms rely on long established network monitoring solutions like Zabbix. As such solutions depend on a seperate monitoring server and communication strategy, the impose some amount of overhead on the platform and make the deployment and configuration more difficult.

To make such information available without causing extra overhead, the proposed monitoring subsystem provides an integrated monitoring solution that allows to extend the information gathered via LLDP with other useful information like CPU utilization and the amount of free memory. This is possible by inserting an additional probing step between parsing the LLDP packet and forwarding its contained latency information to the NFVO as shown in figure 4.8.

The PoC implementation contains several probes for measuring CPU utilization and available memory.

This CPU information is calculated by the values the Linux kernel provides in the `/proc/stat`. According to the Kernel documentation [24], the file `/proc/stat` contains a list of values representing the time the CPU has spend in each available category. These categories can be roughly sperated in user mode, kernel mode and idle time. Without consideration of the time unit of these values, the division of the time spent in all categories except idle time by the sum of all categories yields the CPU utilization. Without further processing, this calculation yields the mean CPU utilization since system startup. For a differet time period, the underlying values have to be subtracted by the values at a given reference point first. The available memory is directly read from `/proc/meminfo`.

In addition to this, the PoC also features Cgroup aware probes for measuring the aforementioned values. Without going into further detail on Cgroup (which is outside the scope of this work), this mechanism provided by the Linux kernel is used to restrict the CPU time for a set of processes. This is done by specifying the value of CPU quota per CPU period. Dividing quota by period, the resulting value represents the amount of CPU power, a process can use. A value of 1 refers to 100 % on a single core, regardless of the number of cores the systems provides.

When the Cgroup feature *CPU accounting* is activated, the Linux kernel provides the amount of time in nanoseconds the CPU spent in a given *slice*[4] in the file `/sys/fs/cgroup/SLICE/cpu,cpuacct/cpuacct.usage`. With this information, the CPU utilization of Cgroup slices is calculated by equation 4.7.

$$CpuUtilization = \frac{CgroupUsage}{\Delta t} \cdot \frac{CgroupPeriod}{CgroupQuota} \tag{4.7}$$

The amount of available memory is calculated by subtracting the given memory limit by the sum of values found in `/sys/fs/cgroup/memory/memory.stat` for the individual categories of memory usage. Examples for such categories are the space required for caching data and for storing the actual program code of a process in memory. The latter is often referenced to under the name of Resident Set Size (RSS) memory.

4.4 Placement Subsystem

4.4.1 Network Model

The information gathered with the monitoring subsystem are used by the NFVO to build a comprehensive network model. The concept of such a network model is to create a directed graph representing the network topology and storing all measured values as edge and node attributes of that graph.

In case of the PoC implementation, each edge in the directed graph has attributes for the destination MAC address, it input port number of the destination switch and the forwarding latency.

[4] A definition of Cgroup limits that can be applied on multiple processes

Each node features the attributes node ID and node type. The node type is a numerical value for distinguishing switches from workers. In case of a switch, the node ID is simply the Data Path ID (DPID) of the switch as assigned by the NFVO in its role als SDN controller of the underlay network. In case of a worker, the node ID is a Universally Unique Identifier (UUID) generated by the NFVM. This is a unique random value with a size of 16 byte that is used to identify the individual workers in the network. Using DPIDs for workers is not possible as the cooperation of NFVO and NFVM is not following the OpenFlow protocol as it is the case for switches.

4.4.2 Service Model

As mentioned before, a network service if modeled by the platform as a Service Function Chain (SFC). The model of a SFC features the attributes used for creating the aforementioned service hooks. In case of the PoC implementation, this is simply the combination of the source and destination IP address but could generally comprise more attributes than that.

In addition to the matching attributes, a SFC has attributes for specifying the placement strategy, a timeout value used for automatic removal and an *immediate* flag that controls wheter all VNFs of the chain should be started immediately after the service hook was hit by a packet, or as late as possible, when the packet is arriving at the worker.

The VNFs to be deployed are referencend in a list of *jobs* that represents the set of operations to be executed on packets traversing the chain. In case of the PoC, where all VNFs are assumed to be containerized applications managed by the Docker engine, the VNF reference in the list of jobs is simply the container name to be used as VNF.

The Internet Engineering Task Force (IETF) proposed a Network Service Header (NSH) in *RFC 8300* [22, sec. 2.3] in which the combination of a Service Path Identifier (SPI) and a Service Index (SI) is used for routing a packet along the path of a deployed SFC. In that regard, the SPI would be an ID common to all stations in the chain while the SI would be used in a similar way as a TTL to keep track of the progress of a packet along its traversal of the network service.

The approach presented in [22] does not take the current limitations of OpenFlow into account (as it is not in the scope of such proposals) which currently does not feature any support for encapsulating packets with NSH.

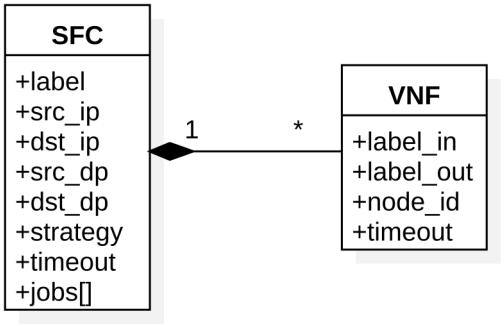

Figure 4.9: SFC model containing the attributes used for service embedding

As an alternative to a regular NSH, this work uses Multiprotocol Label Switching (MPLS) labels for the routing of packets along a deployed service path. Figure 4.9 shows how these labels are stored in the model.

Each VNF has an attribute for storing the label of incoming and outgoing packets respectively. With the label for incoming packets, the VNF itself is addressed. The label for outgoing packets is either a reference to the next VNF in the chain or the egress point of the service. The first VNFs label is also used as an ID for the whole SFC and will be used as the first label pushed on packets hitting the service hook. In that sense, each connection between two stations in the network service, regardless of the type of those stations being VNFs or in- and egress points, is assigned a unique label that is used for routing packets along the given service path.

This concept can be natively deployed on OpenFlow based systems with the optimized data flow presented in figures 4.3 and 4.6. It furthermore has the advantage over the NSH proposed in [22] of being more flexible as it easily allows a *servie header aware* network function (using some kind of Element Management System (EMS)) to dynamically change the label of the next VNF to be routed depending on its outcome or some load balancing strategy. In contrast to this, the NSH envisioned in [22] is designed for a linear traveral of the service path.

4.4.3 Latency Aware Placement

State of the Art NFV platforms do not yet support a latency aware embedding of SFCs. The most important reason for this is their missing ability to measure link latencies of the underlay network. Furthermore, there is also a lack of suitable

algorithms for finding optimal service paths. Even though recent publications like "VNF Placement and Resource Allocation" by Agarwal et al. [25] do propose (and simulate) strategies for an optimized placements of VNFs, such attempts do not seem applicable at the time of writing as they rely on information and capabilites that in some cases cannot be provided by current technologies.

In case of [25], the underlying assumption is that CPU power is quantity that can be compared between different hosts and computing delays on different hosts could be estimated before deployment.

As the provison of CPU power and the benchmarking of VNFs to estimate the incurred computing delay on various hosts is a research topic on its own, this work does only optimize for minimum link latency without regarding the computing delay that would result from a specific deployment.

As a consequence, the PoC implementation of this work does not allow the deployment of more than one VNF of a SFC on an individual worker. The goal of minimizing resource utilization is addressed by incorporating the serverless paradigm and ruling out the deployment of VNFs on hosts whose CPU utilization exceeds a certain limit.

Considering the aforementioned restrictions, a part of the task to build a network softwarization platform for a latency optimized service deployment was to design an algorithm that yields an optimal service path from a source to a destination switch while visiting a certain amount of workers along the way.

Figure 4.10 shows an example of a star topology that is often used in modern cloud computing environments as part of a *spine-leaf* topology. Blue nodes in the diagram are representing switches and yellow nodes are representing workers. In this example, it was the goal to find a route from (1) to (6) while visiting all available workers via the fastest route possible. Please note that this is a directed graph where links incur different latencies depending on the direction taken. The latencies along the links were generated in a random fashion. The link color represents the latency value whereby blue colors represent low latencies and yellow colors high latencies.

The first step of the proposed algorithm is to create a mesh preresentation of the given problem. This is shown on the right side of figure 4.10. The mesh representation only contains the source and destination switch together with the available workers. As this is a fully meshed directed graph, each node has a connection to all

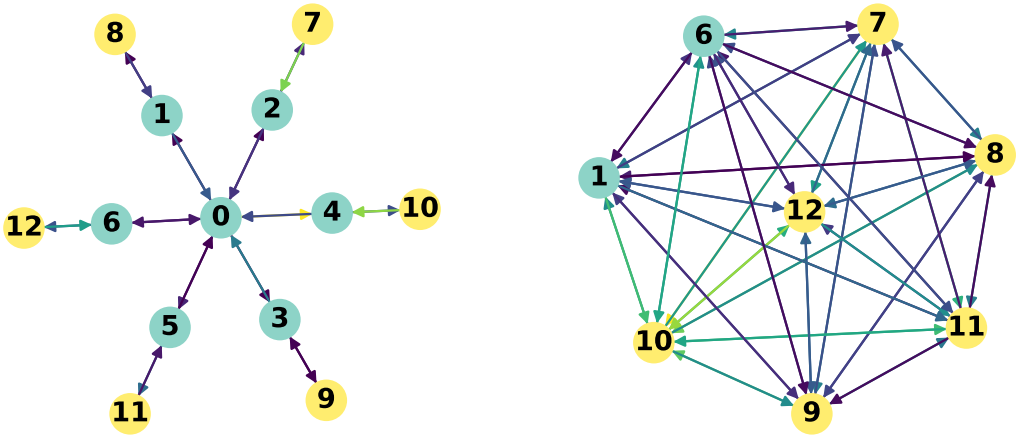

Figure 4.10: The test network (left) used for developing the *Hasty Traveller* algorithm and the mesh representation used for calculating an optimal service path (right)

other nodes. The latencies in the mesh representation have to be calculated with a shortest path algorithm like Dijkstra's or Bellman-Ford. The count of invokations of a shortest path algorithm for creating the mesh representation is expressed in equation 4.8 where N represents the count of available workers including the source and destination switches.

$$C_{SP} = N^2 - N \tag{4.8}$$

After having the mesh representation ready, the core idea of the algorithm is to incrementally visit all (unvisited) neighbors of a node until the path length matches the length of the SFC to be deployed. The sum of all latencies incurred along the path together witch the latencies inccurred by traversing from the last worker to the destination is then used to compare all paths found in that way.

The mesh representation guarantees the minimum amount of invokations of the shortest path algorithm. The operations used for finding an optimal path are visiting the neighbors of a given node and summing up (previously calculated) latencies along the investigated paths.

The aforementioned strategy is in essence a bute-force search of all possible paths and results in a very high number of operations when more than a few available workers are investigated. In the worst case, where the amount of available workers exceeds the chain length to be deployed, the amount of operations is expressed

Algorithm 1 Hasty Traveller (Recursive)

procedure GETPATHS($state, node, level$)
  $paths = []$ ▷ Empty list
  $submesh =$ subgraph of $state.mesh$ for all nodes not in $state.tabu$
  $neighs =$ adjacent nodes of $node$ in $submesh$ sorted by latency
  $costs =$ paths costs (latency) found in $submesh$ for $neighs$
  $state.tabu \leftarrow node$
  for i in $\min(|neighs|, state.limit)$ **do**
    for $cost, path$ in GETPATHS($state$, $neighs[i]$, $level - 1$) **do**
      $paths \leftarrow (cost + costs[i], path + node)$
  $state.tabu \rightarrow node$ ▷ Remove $node$ from $state.tabu$
  return $paths$ ▷ List of tuples ($cost$, $path$)

with equation 4.9, in which n represents the chain length and N is used as before.

The underlying problem of this strategy is known under the term of the *travelling salesman problem*, where an optimal path for a salesman visiting multiple cities is searched. This problem is known to be NP-complete, wherby Nondeterministic Polynomial Time (NP) refers to the fact that with a growing scale of possibilities (in our case an increasing number of workers), the problem complexity is quickly rising to a level where a solution achieved in a reasonable amount of time is highly unlikely. The amount of operations expressed in equation 4.9 confirms this fact.

$$C_{BF} = N! + \sum_{k=1}^{n} \frac{N!}{(N-k)!} \tag{4.9}$$

To introduce a limitation to the complexity described above, the author designed an algorithm called *Hasty Traveller*. This algorithm is a modification of the aforementioned brute-force search in a way that the neighbor count to be visited for searching the best path is limited to m nodes. The amount of operations is therby reduced to:

$$C_{HT} = m^n + \sum_{k=1}^{n} m^k \tag{4.10}$$

It should be obvious that this approach may not yield the optimal solution. To increase the likelihood of a near-optimal solution, neighbors are sorted by increasing latency before being visited for searching optimal paths.

In the PoC implementation, the placement algorithms are implemented in an Ob-

ject Oriented Programming (OOP) fashion. This allows the aforementioned algorithm to store some state in the associated object. Beside other attributes, such a state comprises the mesh representation and a tabu list for keeping track nodes that were already visited before. The *Hasty Traveller* algorithm shown in the depiction above was designed in a recursive fashion. The recursion depth should incur no problem for real world systems as this value coincides with the chain length which is very unlikely to exceed the recursion limit of serveral hundreds on modern systems.

As seen in the problem description for the aforementioned algorithm, it is necessary to know the entry and exit points of a network service in the underlay network. This is the reason why the SFC model in figure 4.9 features the attributes `src_dp` and `dst_dp`. Theses attributes hold the Data Path ID (DPID) of the in- and egress switches as detected with the mechanism introduced in section 4.2.2.

5 Evaluation

5.1 Feasibility of LLDP Looping

The monitoring system proposed in section 4.3 is based on the concept of LLDP Looping that was first introduced in [6], wherein the authors use this concept only for simple RTT measurements.

However, seeing the two entwined round trips in figure 5.1, the question arises how much information can be ultimately extracted from this communication scheme. To answer that question, this section contains an evaluation of LLDP Looping and its feasibility in context of this work.

Figure 5.1 depicts the measurable timestamps (T_1, T_2, T_3, T_4) of basic LLDP Looping. These are expressed with modeling parameters (t_0, t_s, t_r, t_{offs}) in the following way.

$$T_1 = t_0 \tag{5.1}$$

$$T_2 = t_0 + t_s + t_{offs} \tag{5.2}$$

$$T_3 = t_0 + t_s + t_r \tag{5.3}$$

$$T_4 = t_0 + +2t_s + t_r + t_{offs} \tag{5.4}$$

Hereby, the parameters t_s, t_r express the asymmetric latency across the link and t_{offs} refers to the clock offset between the two endpoints. If t_{offs} were zero, the calculation of t_s, t_r would be trivial as the clocks were perfectly comparable. In reality, this is rarely the case.

The modeling equations can be expressed as the following linear system.

$$\underbrace{\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 2 & 1 & 1 \end{pmatrix}}_{det=0,\ rk=2} \cdot \begin{pmatrix} t_s \\ t_r \\ t_{offs} \end{pmatrix} = \begin{pmatrix} T_2 - T_1 \\ T_3 - T_1 \\ T_4 - T_1 \end{pmatrix} \tag{5.5}$$

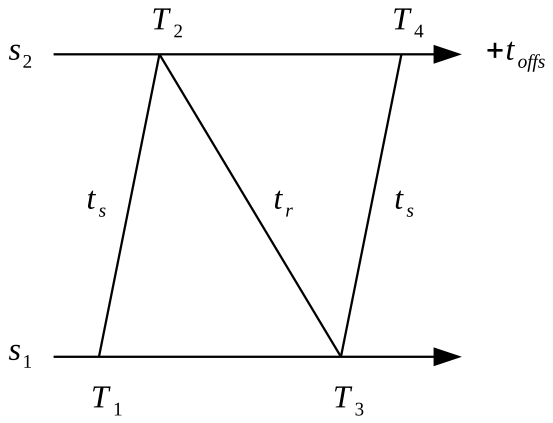

Figure 5.1: Measurable and modeling parameters for basic LLDP Looping

A solution of this system would yield values for the respective latencies as well as the clock offset between the nodes.

Unfortunately, the equations used for modeling the concept are not linearly independent as the third equation in system 5.5 can be expressed as a sum of its predecessors. Furthermore, the determinant of the model matrix is zero, which yields the insight that the above system may have an infinite number of solutions.

Another attempt to model the communication scheme could start with the following equations.

$$T_2 - T_1 = t_s + t_{offs} \tag{5.6}$$

$$T_3 - T_2 = t_s - t_{offs} \tag{5.7}$$

Whereas the third equation could be derived from eq. 5.8 (left) that is used in the NTP for determining the clock offset.

$$\Theta = \frac{(T_2 - T_1) - (T_3 - T_2)}{2} = \frac{(t_s + t_{offs}) - (t_r - t_{offs})}{2} \tag{5.8}$$

$$T_2 - \frac{T_1 + T_3}{2} = t_{offs} + \underbrace{\frac{t_s - t_r}{2}}_{0 \; if \; t_s = t_r} \tag{5.9}$$

But just like for the system assembled before, the determinant and rank values for the model matrix in equation 5.10 are hinting the presence of not linearly indepen-

dent subequations and an infinite number of solutions.

$$\underbrace{\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix}}_{det=0,\, rk=2} \cdot \begin{pmatrix} t_s \\ t_r \\ t_{offs} \end{pmatrix} = \begin{pmatrix} T_2 - T_1 \\ T_3 - T_2 \\ T_2 - \frac{1}{2}(T_1 + T_3) \end{pmatrix} \tag{5.10}$$

This leads to the conclusion that the underlying problem has a higher degree of freedom than can be expressed with the measurable timestamps described above.

It is therefore necessary to restrict the solution space, what could be done in two ways. Either by assuming that t_{offs} is already known, whch would allow the calculation of asymmetric latencies. Or by assuming a symmetric link, which would allow the calculation of the clock offset. The latter case is depicted in equation 5.11.

$$\underbrace{\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}}_{det=1,\, rk=3} \cdot \begin{pmatrix} t_s \\ t_r \\ t_{offs} \end{pmatrix} = \begin{pmatrix} T_2 - T_1 \\ T_3 - T_2 \\ T_2 - \frac{1}{2}(T_1 + T_3) \end{pmatrix} \tag{5.11}$$

The modification of eq. 5.11 compared to eq. 5.10 was introduced by imploring the given assumption on eq. 5.9. For both assumptions, the linear system becomes solveable.

Calculating the model parameters using the linear system above is basically the same as approximating t_s, t_r by using Θ as from equation 5.8.

$$\widehat{t_s} = T_2 - T_1 - \Theta \tag{5.12}$$

$$\widehat{t_r} = T_3 - T_2 + \Theta \tag{5.13}$$

This approximation ultimately leads to the following relative errors if the assumption of a symmetric link is not satisfied.

$$\frac{\widehat{t_s} - t_s}{RTT} = -\frac{t_s - t_r}{2 \cdot RTT} \tag{5.14}$$

$$\frac{\widehat{t_r} - t_r}{RTT} = \frac{t_s - t_r}{2 \cdot RTT} \tag{5.15}$$

It can be seen that in the worst case, where RTT is solely made up by one of the latencies while the other remains zero, the absolute error is $\pm\frac{RTT}{2}$.

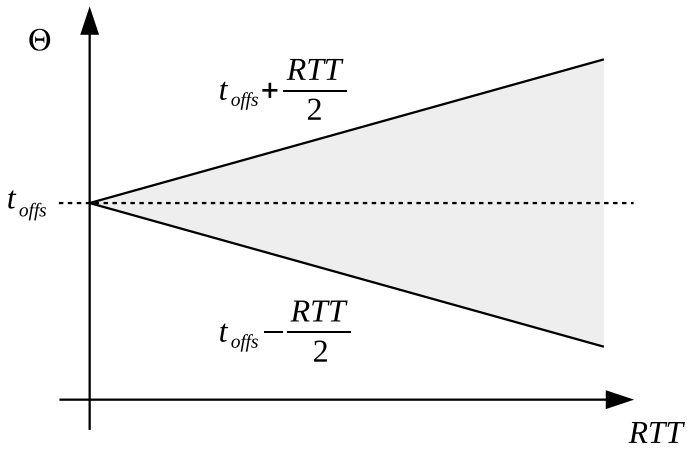

Figure 5.2: Offset-delay plot used for the clock filter algorithm in NTP

Another representation of the maximum errors described above is found in the offset-delay plot shown in figure 5.2 wherein these errors represent the boundary of the measurement space. All measured combinations of RTT and Θ values will lie inside the grey area in figure 5.2. The NTP protocol defines a *clock filter algorithm* that is used to identify good measurements (that lie as close to the leftmost pinnacle of the grey area as possible) for calculating the clock offset with minimal error.

For the monitoring solution proposed in section 4.3, the aforementioned clock filter algorithm could be implemented as part of LLDP Looping as a way to restrict the solution space as mentioned before. This would further extend this monitoring solution with integrated clock synchronization. As an advantage, this would minimize control overhead, as the LLDP based latency measurement and topology detection moduke is running anyway, and nno sperate deployment of NTP or similar clock synchronization protols would necessary.

As an alternative, the calculations above have shown that by assuming a symmetric link, the clock offset can be computed directly. It was already mentioned in section 4.3 that incorporaiting XDP in the monitoring system allows for differntiating between propagation and queueing delays whereby the former is assumed to be generally symmetric. This assumption should be satisfied when the measured link forms a direct connection between the monitored endpoints and is not spanned accros mutliple non-SDN switches.

Reviewing the evaluation above, the combined asymmetric latency measurement

and clock synchronization as suggested by the communcation scheme of LLDP Looping is not directly possible. The measurable parameters simply do not allow to distinguish between time differences induced by clock offsets or asymmetric latencies. However, with the clock filter algorithm known from NTP and the approximation of clock offset via the separation of queuing and propagation delays with XDP, the author has suggested two possible solutions that could be investigated by further research.

5.2 Latency Aware Service Embedding

In order to support a latency aware network service embedding, the *Hasty Traveller* algorithm was developed in section 4.4.3. This algorithm tries to reduce the computing time of a brute-force search of all possible deployments by introducing a limit for the number of paths to be investigated at each possible node in the mesh representation. To increase the likeliness of finding an optimal path, all adjacent neighbors of a node are sorted by link latency before investigating the limited number of paths.

The proposed algorithm was used in figure 5.3 to evaluate the performance of latency aware service embedding in relation to the length of the deployed SFCs and the aforementioned *limit* value. The depicted values are end-to-end service latencies for an embedding of the given SFC in the network model shown in figure 4.10 with the only difference of featuring 7 workers instead of 6.

The dark patch labeled *RP-1000* in figure 5.3 represents the service latencies observed in a set of 1000 random deployments. In front of this, several deployments using Hasty Traveller (HT-x) are depicted whereby the number shown in the legend refers to the limit value used for the path finding algorithm.

It can be seen in figure 5.3 that the algorithm yields optimal solutions when the limit equals the chain length. That makes sense as this is basically a brute-force search for the optimal solution. Furthermore, and quite naturally as well, the figure shows that by increasing the limit value, the algorithm is more likely to yield an optimal solution for all chain lengths. Most noteworthy is the fact that in this example, the relatively low limit value of 3 is already sufficient for finding an optimal placement in most cases.

Whereas figure 5.3 proves the ability of the proposed algorithm to yield latency

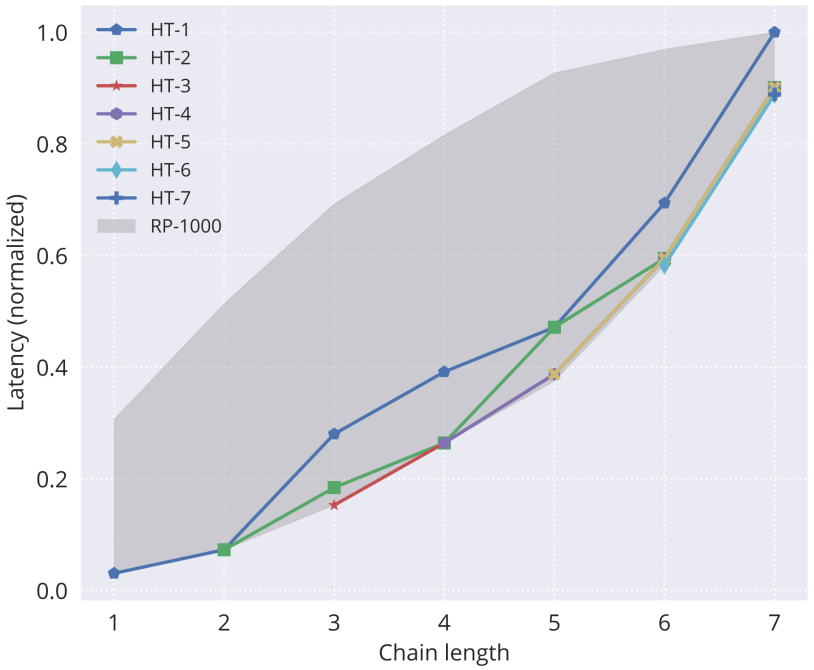

Figure 5.3: Service latency as a function of chain length for various algorithms

optimized placements, the computational complexity for this algorithm shown in figure 5.4. The measured values were computed on a single core of a Intel Core i3-2100 CPU.

On the left side of figure 5.4, it is shown that (near) brute-force searches for feasible lead to extremly fast growing computation times, whereas the limitations introduced in *Hasty Traveller* are able to reduce this complexity by nearly an order of magnitude.

It should be noted that the computational complexity of random placement (or more specifically, of 1000 random placement calculations) is growing linearly with the chain length and introduces a tradeoff between the two placement algorithms beginning with chain lengths greater than 6 in this example.

However, we have already seen in figure 5.3 that the high limit values for those the tradeoff is already visible at relatively short chain lengths should not be used anyway, as they introduce higher computation time without improving the end-to-end latency.

Another interesting aspect is shown on the right side of figure 5.3. In this figure, the

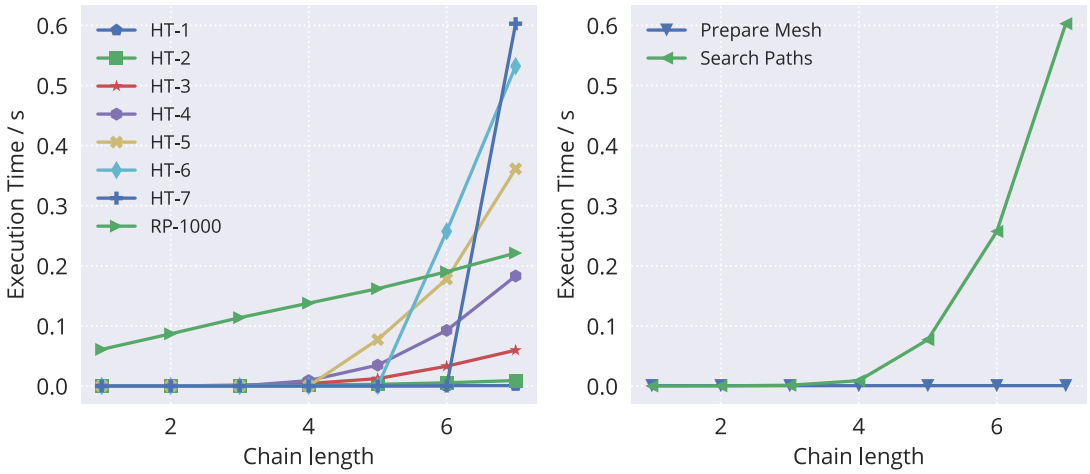

Figure 5.4: Comparison of computation time for various placement strategies

time it takes to generate the mesh representation is compared to the worst case computation time of path search with the (quite surprising) result, that this step of the algorithms is nearly negligible in comparison to the recursive path searching algorithm. A probable reason for this is the use of a rather simple and single-threaded implementation in pure Python for the PoC implementation of the algorithm.

The evalutation of the proposed placement algorithm for a latency aware service embedding has arisen some concerns as to the scalability of the solution. However, choosing relatively low limit values can reduce the computation time greatly and the current implementation has great potential for further optimization. These aspects lead to the conclusion that *Hasty Traveller* is indeed a feasible placement algorithm to be used in the context of future NFV platforms.

6 Conclusion

6.1 Review

Conventional network softwarization platforms are often build on top of virtualized cloud infrastructure platforms like OpenStack and therefore have to cope with the abstracted view of the underlying infrastructure such platforms typically provide.

This is a prolific cooperation as long as the development of NFV platforms are occupied with business related challenges like providing resilient authentication, management and billing solutions mostly facing the platforms users or clients.

As a downside of being dependent on these rather complex systems, the implementation of new features and paradigms became ever more difficult for researchers on the field of NFV.

Combining these circumstances with the recent advances in communication technology like 5G that is envisioned to enable many new use cases that need an agile environment that is able to adapt itself to changing parameters, the lack of lightweight network softwarization platforms for quick prototyping and easy deployment becomes increasingly evident. This work made an attempt to fill that gap.

Having reviewed the state of the art NFV platforms and standardization efforts, the author identified the lacking monitoring capabilities of conventional platforms as a main problem hindering the development of more agile NFV platforms. Being based on regular cloud infrastructure platforms like OpenStack, state of the art NFV solution lack the awareness of performance metrics of the underlay network.

This work provides a solution to mitigate this by proposing an integrated network monitoring system for NFV platforms that is able to measure important parameters like link latency and resource utilization of available workers without the need for further administration. This monitoring system does furthermore allow automatic topology and compute node detection.

Whereas the automatic duplication of already deployed VNFs is already possible in state of the art solutions, it is still an open problem to reduce the amount of administration that is necessary for VNF deployment.

To that end, the author developed an optimized data flow and SFC lifecycle management following the emerging paradigm of serverless computing. As no generally formulated algorithms for providing a latency-optimized service function embedding that fit the developed subsystems were found in the state of the art, the author to that purpose designed a new algorithm called *Hasty Traveller* that drastically limits the computation time necessary for SFC embedding when compared with a brute-force search. It is shown in the evaluation chapter that this algorithm is able to support a latency-optimized placement of VNFs.

The concepts and systems mentioned above were furthermore implemented in a way to enable emulation and testing on commodity hardware. By providing the in-depth architectorial description in chapter 4 together with a PoC implementation, the author hopes to have fulfilled to goal to shep some light in the vast gap between high level mathematical modeling done by researchers on this field and state of the art implementations development in the associated industry.

6.2 Outlook & Future Directions

As this was an initial work on building a network softwarization platform following the serverless paradigm for managing services and VNFs, there is plenty of research oppurtunity left over.

Most importantly, it is still an open question how to reliably compare and assign computing power on different host (that may be based on different hardware as well). In that regard, also a solution for VNFs benchmarking would be desirable.

In the field of monitoring solutions, the task of incorporating clock synchronization into the monitoring platform to reduce control overhead is still open. It would also be interesting to investigate or develop other synchronization algorithms beside the well known clock filter algorithm used by NTP.

For Management and Orchestration (MANO), the scaling capabilities could be enhanced further. For example the a solution on how to incorporate automatic VNF duplication and load balancing could be investigated.

And in a more general sense, the proposed platform could also be used for design-

ing new and improves placement algorithms beside the *Hasty Traveller* proposed in this work. For such use cases, the developed platform should provide a solution for reproducible experimentation as the range of users and (business) requirements for such a lightweight platform is limited.

Bibliography

[1] ONF, "Sdn architecture overview," tech. rep., Open Networking Foundation, Dec. 2013.

[2] J. M. Watkins, "Openflow based traffic engineering for mobile devices," 2014.

[3] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and latency of virtual switching with open vswitch: A quantitative analysis," *Journal of Network and Systems Management*, vol. 26, 07 2017.

[4] S. K.Mohapatra, J. Bhuyan, and H. N.Narang, "Planning and managing virtualized next generation networks," *International journal of Computer Networks & Communications*, vol. 7, pp. 01–16, 11 2015.

[5] OpenBaton, "The openbaton online documentation." `https://openbaton-docs.readthedocs.io/en/4.0.0/`. Online, accessed on September 3, 2019.

[6] L. Liao, V. Leung, and M. Chen, "An efficient and accurate link latency monitoring method for low-latency software-defined networks," *IEEE Transactions on Instrumentation and Measurement*, vol. PP, pp. 1–15, 07 2018.

[7] M. Condoluci and T. Mahmoodi, "Softwarization and virtualization in 5g mobile networks: Benefits, trends and challenges," *Computer Networks*, vol. 146, 09 2018.

[8] B. Blanco, J. O. Fajardo, I. Giannoulakis, E. Kafetzakis, S. Peng, J. Pérez-Romero, I. Trajkovska, P. Sayyad Khodashenas, L. Goratti, M. Paolino, and E. Sfakianakis, "Technology pillars in the architecture of future 5g mobile networks: Nfv, mec and sdn," *Computer Standards & Interfaces*, vol. 54, 01 2017.

[9] ONF, "Software-defined networking: The new norm for networks," tech. rep., Open Networking Foundation, April 2012.

[10] ONF, "Openflow switch specification," tech. rep., Open Networking Foundation, Mar. 2015.

[11] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 117–130, USENIX Association, May 2015.

[12] W. Tu, J. Stringer, Y. Sun, and Y.-H. Wei, "Bringing the power of ebpf to open vswitch," in *Linux Plumbers Conference*, 2018.

[13] ETSI, "Network functions virtualisation (nfv)." `https://www.etsi.org/technologies/nfv`. Online, accessed on September 21, 2019.

[14] ETSI, "Osm scope, functionality, operation and integration guidelines," tech. rep., Feb. 2019.

[15] K. Phemius and M. Bouet, "Monitoring latency with openflow," pp. 122–125, 10 2013.

[16] L. Liao and V. Leung, "Lldp based link latency monitoring in software defined networks," pp. 330–335, 10 2016.

[17] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), pp. 133–146, USENIX Association, 2018.

[18] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," pp. 405–410, 06 2017.

[19] P. Aditya, I. Ekin Akkus, A. Beck, R. Chen, V. Hilt, I. Rimac, K. Satzke, and M. Stein, "Will serverless computing revolutionize nfv?," *Proceedings of the IEEE*, vol. PP, pp. 1–12, 02 2019.

[20] C. Sarathchandra, D. Trossen, and M. Boniface, "In-network computing for app-centric micro-services." `https://www.ietf.org/id/draft-sarathchandra-coin-appcentres-00.txt`. Online, accessed on October 9, 2019.

[21] T. Høiland-Jørgensen, J. Dangaard Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: fast programmable packet processing in the operating system kernel," pp. 54–66, 12 2018.

[22] P. Quinn, U. Elzur, and C. Pignataro, "Network service header (nsh)," RFC 8300, RFC Editor, January 2018.

[23] L. L. Peterson and B. S. Davie, *Computer Networks, Fifth Edition: A Systems Ap-*

proach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[24] T. Bowden, B. Bauer, J. Nerin, S. Feng, and S. Seibold, "The /proc filesystem." `https://www.kernel.org/doc/Documentation/filesystems/proc.txt`, 2009. Online, accessed on September 22, 2019.

[25] S. Agarwal, F. Malandrino, C.-F. Chiasserini, and S. De, "Vnf placement and resource allocation for the support of vertical services in 5g networks," *IEEE/ACM Transactions on Networking*, vol. PP, 12 2018.