Aalborg University Copenhagen
A.C. Meyers Vænge 15
2450 København SV

Secretary: Maiken Keller

AALBORG UNIVERSITY
STUDENT REPORT

Semester:
ICTE 4

Title:
Distributed Software-Defined Networking Emulation

Project Period:
Spring Semester 2019

Semester Theme:
Master thesis

Supervisor(s):
Henning Olesen

Project group no.:
4SER 4.3

Member(s):
20145350 - Lukas Buch Jordell

Pages: 93

Finished:
June 6, 2019

Abstract:

Software-Defined Networking is an emergent field within networking, which takes a new approach to network management over traditional networks, by abstract control logic from the forwarding functions of network functions. The migration between the two are not withstanding issues and challenges relating to the migration of Operating Support Systems, Re-factoring application architectures and training and technological implementation surrounding SDN components.

In this project, the concept of network emulation is introduced, and a distributed emulated testbed is proposed. The proposed prototype seeks solve for the stipulated issues and challenges, by enabling developers to implement SDN applications and test and monitor SDN controller and OpenFlow interaction and configurations, by providing a dynamically provisioned and emulated data layer that these may manage.

Distributed Software-Defined Networking Emulation

Master Thesis

Lukas Buch Jordell

Aalborg University
Innovative Communication Technologies and Entrepreneurship

Contents

List of Figures

Chapter 1

Introduction

1.1 Research Motivation

Information and Communication Technologies (ICT) profoundly impact and permeate multiple aspects of modern day life, such as facilitating the convergence and transformation of social, economic and educational sectors, as well as creating new divergent markets. Meanwhile, inter-connectivity is growing at an exponential rate with a continued shift in the number and variants of devices and connections, as the world is moving towards the implementation of fifth generation (5G) network technologies [1].

Here, IPv6 and Low-Power Wide Area Network (LPWAN), among others, are technologies enabling Internet-of-Thing (IoT) connectivity and machine-to-machine (M2M) applications across industries [2], further accelerating IoT growth [3]. A development that is occurring alongside broadband developments [4] and an escalating demand for various resource intensive services, e.g. asymmetric application traffic growth associated with the advent of Over-The-Top (OTT) multimedia Content Delivery Networks (CDN), cloud-based platforms and application services, e.g. Google, Facebook and Amazon [5]. Within the area of operational sustainability (CAPEX/OPEX) and service reliability (QoS/QoE), there is an objective to minimize the Total Cost of Ownership (TCO) of network infrastructure and Return Of Investment (ROI).

Many concurrent traditional Transmission Control Protocol/Internet Protocol (TCP/IP)-based internet architectures, have not been in a state to effectively meet these drastically developing demands with its established best-effort end-to-end communications model between network endpoints [6]. Moreover, the Internet architecture has become complex as networks vary significantly in the respective capabilities and functions provided. This is exacerbated, as each implement varied proprietary hardware, software and middle-boxes, e.g. application-specific gateways or proxy servers, necessary to cater to newer and more demanding applications supported by modern systems [7][8]. A development furthered by contending market actors, e.g. Communication Service Providers (CSP) navigating multi-vendor environments. All of these contribute to a static and complex state of operation, a

phenomenon known as Internet ossification [9].

This outlined situation has been leading to vendor lock-in, complexity and lack of flexibility related to core provisioning of new network components on various scales [9]. The resulting outcome thereof is the introduction of higher barriers to entry for insurgent market actors, whilst further constituting a hindrance to the adoption of new and disruptive network technologies and services for incumbent market actors. Situations that can be debated as being inherent obstacles in enabling innovation in the provisioning and management of the respective networks, in order to meet increasing business and network requirements [7].

To address such challenges, CSPs started to embrace the concept of network abstractions through virtualization of their respective networks. This trend is largely inspired by the overwhelming success of virtualization within the field of cloud computing, where it addresses similar challenges [10]. Virtualizationn is the key enabler for various cloud-based service models, e.g. Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) [10]. These, and orchestration solutions, such as Kubernetes [11], OpenShift [12], Docker [13] and OpenStack [14], have largely lead to the proliferation and re-factoring of application service orchestration through innovation in provisioning and management of virtual resources and service elements, e.g. computer, storage and network over underlying Commercial Off-The-Shelf (COTS) hardware [10][3]. This have hailed the logical evolution from Monolithic architectures to Service-Oriented Architectures (SOA) to Micro-Service Architectures (MSA) [15]. A change that can be observed in many enterprises and research articles, where containers and virtual machines are being employed to host applications and virtual network functions.

Networks are typically constituted by a larger number of computational networking nodes and storage devices. These components are the network service functions associated with routers, switches, firewalls, load balancers, Domain Name System (DNS) servers and other dedicated network equipment in core and access networks, e.g. customer premises equipment [4][3]. Displayed in figure 1.1 is a representative high-level overview of the different networking domains that may be associated with a typical CSP network. Here, CSPs are moving to virtualize network functions to enable faster deployment and scalability of the constituent networks [16].

As a step towards virtualization, most CSPs have implemented overlay networks based on a series encapsulation techniques, which use tunneling, tagging, and labeling protocols, e.g. Generic Routing Encapsulation (GRE), Point-to-Point Tunneling Protocol (PPTP), Layer 2 Tunneling Protocol (L2TP), Virtual Extensible Local Area Network (VXLAN) and Multiprotocol Label Switching (MPLS), in order to enhance software control over network functions. While this approach enables innovation by distinct network services, e.g. Virtual Private Networks (VPNs), it does not change the underlying infrastructure characteristics. Thus, it realizes only the short-term benefits of network virtualization [3]. To realize the potential of network virtualization in solving stated challenges, there is a necessity in enabling the virtualization of network functions over whitebox hardware and software [3].

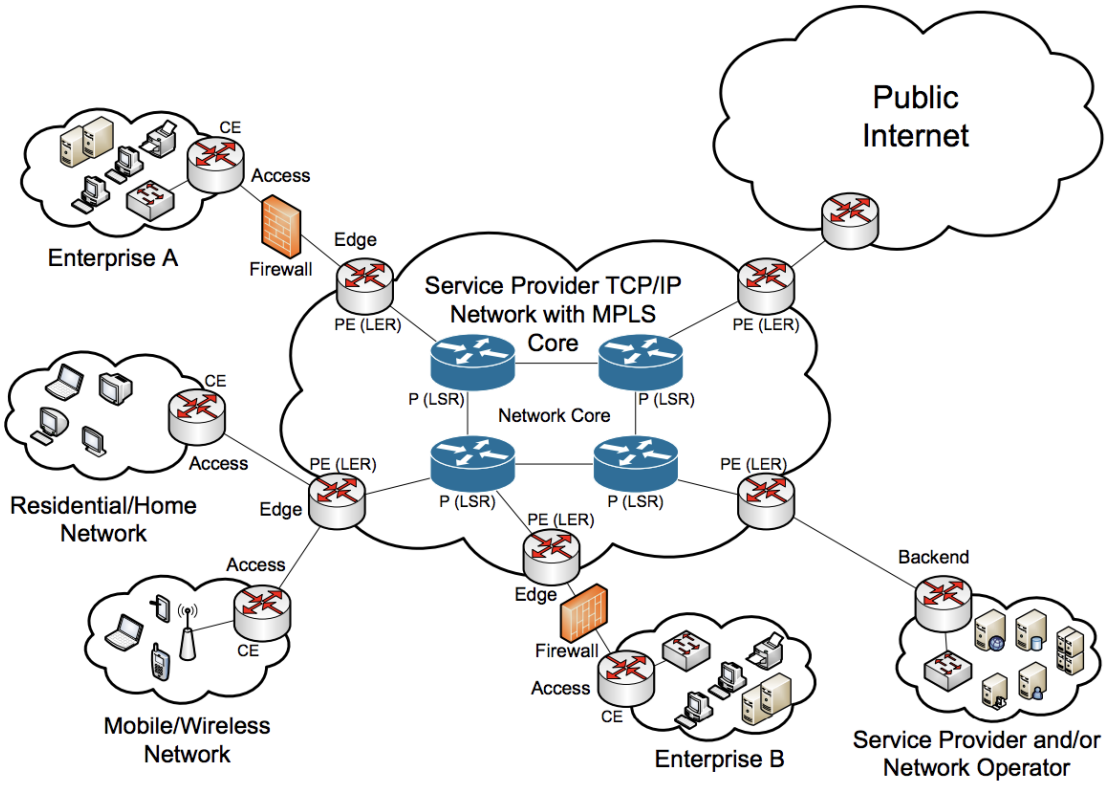

Figure 1.1: Representative Communication Service Provider core and access network facilities [17]

With that in mind, more recent developments within the field of network development and management, are displaying new trends and approaches to network virtualization and management. Among these approaches, Software-Defined Networking (SDN) is at the forefront of concurrent research topics and application in the networking community. A technology that have a potential to significantly impact operational sustainability [18][19].

With regards to the high-level overview of the SDN reference architecture illustrated in figure 1.2, the core concepts behind SDN technological approach to network management is the software abstraction of control logic from the forwarding mechanisms of network functions. By decoupling the data forwarding mechanisms of the data layer from the control mechanisms of the control layer, the network functions can become directly programmable. This allows traffic flows to be adjusted from a central logical software controller, thus simplifying network-wide operations. The SDN controller maintains a view of the network state and may alter forwarding function using the OpenFlow protocol [20], in order to meet changing network requirements. In the data layer, the network elements may be comprised of both hardware- and software-based network functions, e.g. core and edge switches presented in figure 1.1. These network functions can be programatically modified via SDN applications in the application layer, relating to the automation,

configuration and management, monitoring and troubleshooting of network resources, in order to provide end-to-end solutions. Because the SDN architecture is based on open source software, open standards and a central logical controller, it appears more like cloud computing than traditional network management. It addresses many of the challenges described in the above sections [21].

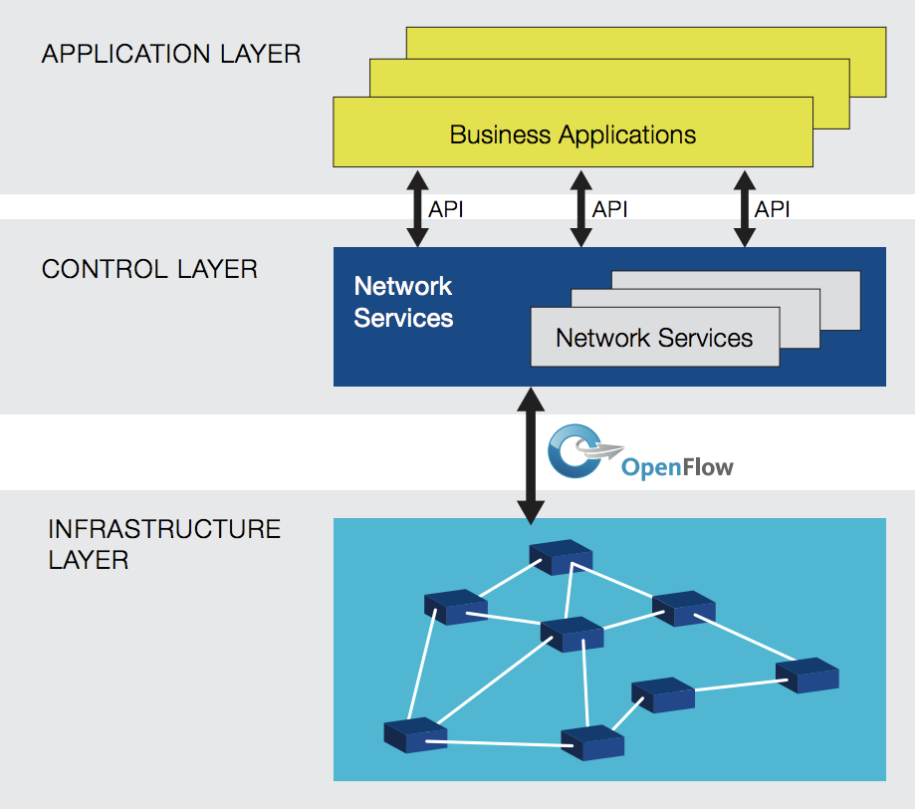

Figure 1.2: A simplified overview of the SDN reference architecture [21]

Dynamic Service Function Chaining (SFC) deployment and provisioning of network functions achievable through the centralized logical control functions of SDN and proliferation of cloud-based execution and automation environments, i.e. Infrasture-as-a-Servce (IaaS) and Platform-as-a-Service (PaaS), have enabled new opportunities regarding these topics. Particularly prominent SDN use cases feature linking physical hardware and virtual network functions, providing virtual service for carrier networks, e.g. virtual Evolved Packet Core (vEPC) and deploying virtual customer edges, i.e. virtual Customer Premises Equipment (vCPE) [16] - refer to figure 1.1.

This development lines up both challenges and opportunities as applications grow in complexity, compelling CSPs to innovate their business models in regards to their available network resources and service components. Network virtualization and programmability

delivers flexibility over existing legacy architectures that can enable the aforementioned service reliability and operational sustainability. Arguably an important objective, standing on the verge of the 5G era [16].

Moving to SDN-enabled networks is not withstanding certain challenges, i.e. converting from traditional systems to software-defined and virtualized systems involves the restructuring of existing architectures and networking environments. In the case of Motorola, refer to appended interview, this involves re-factoring of enterprise applications into a Micro-Service Architecture and considerations on the functional modelling of orchestration systems, i.e. Kubernetes [11] and OpenShift [12] and constituent SDN controllers, i.e. Cilium [22] and Juniper contrail [23][24].

As such, it can be stipulated that a number of challenges must be confronted by CSPs along that process, including decisions on personnel training, performance modelling and evaluation of orchestration systems and SDN controllers, service availability and reliability, vendor interoperatibility and more. Furthermore, in many cases, SDN architectural components must interact with legacy components introducing further complexity [25][26].

1.2 Problem Space

During the preliminary research phase, it was found that there is a general consensus among standard organizations, specification organizations, open source organizations and commercial vendors that SDN solves for challenges posed by increasing internet ossification. Progress achievable through key attributes such are programmability, openness, heterogeneity, and maintainability [25]. It has further been recognized as a key enabling component in future 5G and IoT networks, by organizations such as The 5G Infrastructure Public Private Partnership (5GPPP) [1], Next Generation Mobile Networks (NGMN) [27], Groupe Spéciale Mobile Association (GSMA) [28], European Telecommunications Standards Institute (ETSI) [29], Open Networking Foundation (ONF) and the Broadband Forum (BF) [26]. All of the above sees the standardization of technology as essential for the global success of future solutions and ecosystems.

However, it is also recognized that SDN have yet to reach standardized maturity, which enables the multi-vendor interoperatiblity, economies of scale and clear interfaces needed for SDN to be widely adopted [27]. In a report by Cartesian, made in collaboration with the Broadband Forum (BF), several immediate issues and long term challenges has been identified as an obstruction for widespread implementation. Technological, operational and organizational issues and challenges that needs to be addressed [26][25]. The primary findings show that:

- **Technological** - While many CSPs does want to migrate to SDN, it has been widely acknowledged that the technologies are still in their infancy. This is in part targeted at concerns surrounding reliability and interoperatibility of existing ETSI and ONF specifications, in regards to integration issues tied into existing vendor and legacy

domains, e.g. lack of concise boundary functions. Further in part by the multiple standards, specifications and open source organizations [30] and initiatives and vendors leading to software incompatibilities [26].

- **Operational** - The operational migration from legacy to hybrid to open virtual systems require a significant resource investment, as complexity and requirements is added for CSPs, e.g. new orchestration and service management, multi-vendor environments and training needs to be taken into account. As a result, updating legacy processes and systems to integrate with SDN is causing a bottleneck [26].
- **Organizational** - Digital transformation strategies can in of itself not be solely focused on technological implementations, i.e. organizations are posed with challenges in regards to industry-wide knowledge and practical experience scarcity [26].

The report in mention concludes that there are significant benefits, but also challenges to overcome for CSPs in migrating to open SDN systems. It further proposes a outlines a series of recommendations, supported by ONF [25][31], which CSPs can follow to ease the transitional period between system states in production environments, as it is prospective that it will constitute an extended period of time [26].

Among these recommendations are considerations towards creating a comprehensive strategic plan for SDN deployment in existing architectures and ecosystems, as Operations Support Systems (OSS)/ Business Support System (BSS) and service models needs to be reconsidered [26]. In short network transformation should happen by acquiring the necessary proficiency and resources to do so [26], e.g. functional change may be performed in phases [25][31].

Research and interviews have indicated that it is common practice for SDN controllers to come packaged or as plug-ins in orchestration platforms, such as Kubernetes, OpenShift, Mesos [32], etc., which provide functional models for network operation and orchestration, where policy engines may be employed to achieve a desired service. However, the inner workings and performance models of SDN controllers are often not exposed.

This is supported by an emergent variety of network emulators that have been developed to aid the research community and CSPs in provisioning, testing and analyzing potential network deployments and service models [33][34]. As mentioned, within these there are two perspectives that should be considered. These are the functional and performance models associated with implementing SDN orchestration and architectures.

It is then evident that there are challenges and that these are multi-faceted, e.g. may entail the process of transition, vendor compatibility for software and hardware, network management for rapid traffic growth rates, standard maturation, insufficient deployment knowledge and expertise, linking OSS/BSS with virtual networks and more [26][35].

This poses an opportunity for the development of an emulated testbed that may be employed as a development environment for the planning of network architectures, the implementation and testing of SDN controller and OpenFlow mechanisms and development of SDN applications, before these are potentially deployed into a production environment.

1.3 Research Objectives

Therefore, the main research objective for this thesis becomes the realization of an emulated testbed. The focus is on enabling the provisioning a distributed emulated network that can constitute the data layer for implementing SDN component functionality, i.e. SDN controller and SDN application. Ultimately, this is to assist in solving potential network ossification issues, through innovation and testing of these components in an emulated environment. From this main objective the following objectives are derived:

- Identify the related work and research in the field of networking relating to SDN, network protocols technologies and implementations.
- To Identify the relevant network technologies and architectures and compare these to each other.
- To propose an implementation approach and its methodology to realize the research question for this thesis.

1.4 Research Questions

Based on preliminary research and stipulated research objectives, the following main research question can be deduced:

- ***How can network emulation and monitoring technologies be employed to investigate the interaction between data and control layers components in an emulated SDN architecture?***

In order to answer this research questions, sub-questions have been formulated and sought to be discussed throughout the report. Each sub-questions is formulated in perspective to primary research questions and seek to provide insight into the underlying problems. From the primary research questions, the following sub-questions can be deduced:

- *How may network emulation technologies be employed to provision a network topology?*
- *How can network control and monitoring be implemented in an emulated testbed?*
- *How may network control and data flow metrics be displayed to the user?*
- *How can a provisioned network topology be visualized?*

1.5 Research Scope

In order to delimit the scope of the thesis, some limitations have been stipulated for the project. These are set out as the fields of Software-Defined Networking are considerable and continuously evolving. These limitations include:

- The thesis will be focusing on the development a emulated testbed that can provision isolated and distributed network functions.
- The primary purpose of the emulated testbed is to observe SDN controller and OpenFlow interactions in a switching fabric with associated end-host and subsequent performance modelling.
- Discussing performance modelling, the thesis does not focus on how to incorporated functional modelling.
- A full analysis of all possible integrated SDN and NFV architectures in research and commercial settings will no be investigated.
- Closed sourced or black box solutions will not be taken into consideration due to operational uncertainty. In line with the SDN reference architecture, the proof of concept prototype will solely be based on open-source code and vendor-neutral protocols.
- A limited proof of concept prototype will be implemented, with additional reflections on functionality regarding the future perspective of the prototype.

1.6 Expected outcome

This section outlines the expected outcome of the master thesis project. It contains the envisioned preliminary design of the proposed proof of concept prototype. The prototype will be reflected upon in chapter 6.

As delineated in prior sections, the expected outcome of the thesis project is the development of a network emulation testbed. A tool that can be employed for rapid and dynamic provisioning of an emulated network topology without the need for networking hardware. An emulated network that developers can work on independently and share between them, in order to perform testing and monitoring of SDN controller and OpenFlow interactions. The Proof prototype is comprised of seven functional objectives:

- The prototype should be able to provision singular or multiple connected emulated network topologies.
- The prototype should be able to host network emulators on virtual environments.

- The prototype should be able to provision standard or customized network templates from repositories to recreate network topologies.
- The prototype should be able to monitor network traffic and SDN controller functions and behaviour.
- The prototype should be able to visualize a provisioned network topology and display monitored network metrics
- The prototype should be able to perform performance modelling to observe SDN controller and OpenFlow interactions.
- The network provisioned by the prototype should be able to be managed through a Command-Line Interface, but may be supported by Graphical User Interface (GUI) functionality.

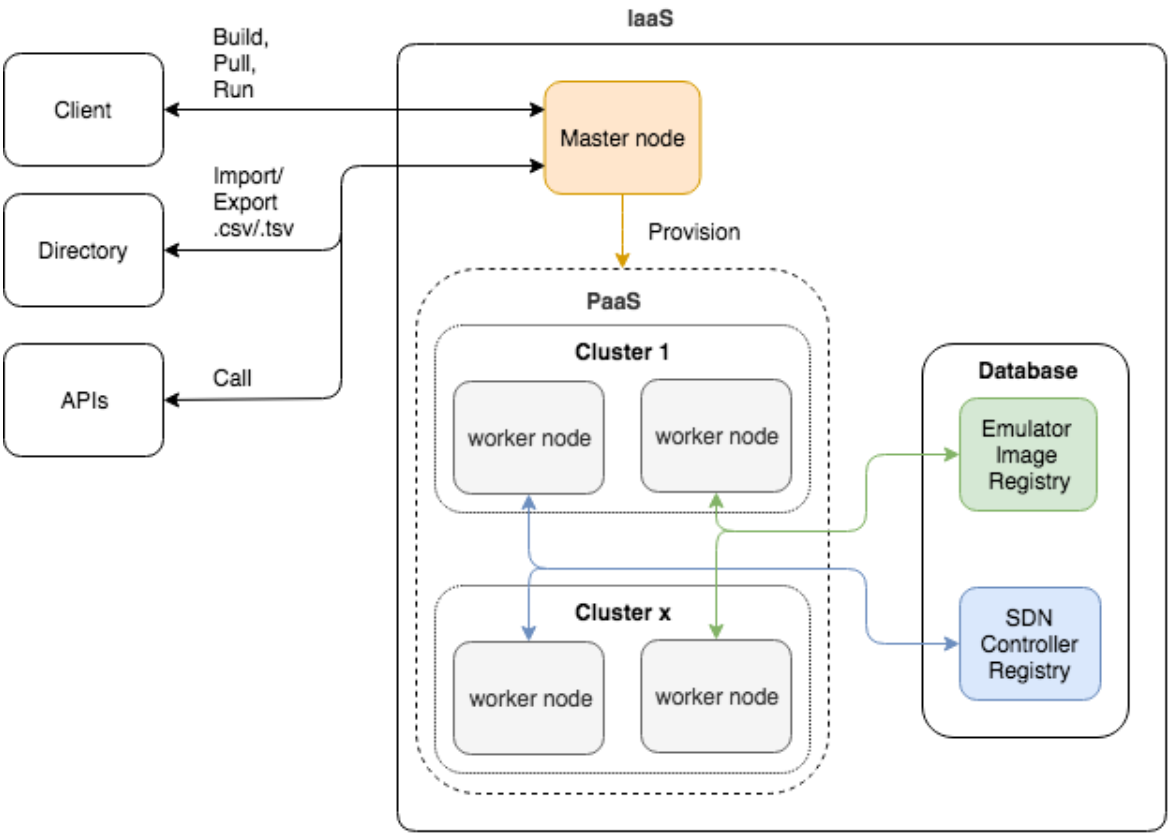

Figure 1.3: Expected outcome of the project and preliminary architecture

To realize this, the proof of concept prototype should be able to be hosted on a IaaS, where a master node will handle incoming client requests. This master should further

handle the subsequent provisioning of virtual environments, e.g. containers or Virtual Machines (VMs) from a constituted provider. Additionally, the master node may handle the allocation of SDN controller and network templates and OS images. As these are changed, they should be saved to the system for allocation at a later date, e.g. if a network worker node is deleted.

The system presented to the client will be comprised of a GUI dashboard and a CLI that should entail the provisioning and allocation of network worker nodes per the above-mentioned functionality. It should further be able to visualize currently operational network worker nodes and network topologies. The client should be able to program the network nodes to construct specific network scenarios. A preliminary system architecture is depicted in figure 1.3.

1.7 Methodology

This section introduces the approach taken and the methods used, in order to gather data and develop a proof of concept prototype that attempts to solve for the questions introduced in section 1.4. The research techniques employed in the data gathering process are detailed together with the project process model.

The primary motivation behind the thesis originates from extended interests within the field of routing and switching and how the lines between software development and network engineering is becoming increasingly blurred. This thesis aims at making progress in the direction of the delineated State-of-the-Art research within the field of networking, which is presented and discussed in Chapters 2 through 6.

1.7.1 Research

The immediate approach to the topic was to employ research methods that are suited for exploring new and emergent areas within the networking discipline. To this extend an exploratory research paradigm was employed to study the problem space thoroughly, as to establish methodological priorities in the shape of research questions - Research question intended to determine the scope of the thesis together with the imposed limitations delineated in section 1.5 and section 6.2.

For this purpose the thesis, the approach predominately relied on secondary research. To elicit the relevant evidence to answer the research questions, a set of sources must be selected to perform the search. Therefore, the following web search engines was selected: ACM, Digital Library, IEEE Xplore, Science Direct, Springer Links and more. A set of keywords and filter criteria was employed in this search to determine content and content types. This search was supplemented by whitepapers and specifications authored by industry organizations, e.g. ONF and ETSI and market leading actors, e.g. Cisco and Juniper. This was carried out in order to determine the applied technological processes necessary to construct a conceptual design and answer specified research questions.

1.7.2 Process model

The thesis project adopts an agile methodological approach to developing the project - An overview is displayed in figure 1.4. The approach borrows several elements from SCRUM method [36]. The approach is centered around exploring and analyzing the State-of-the-Art field associated with the problem space.

As is experienced with prior projects, there will be overlapping research and design phases, as the analysis of technologies or new insight provided by industry interviews, necessitates additional investigation into technologies, code, studies, etc.. This approach is largely in line with the agile methodological ideology.

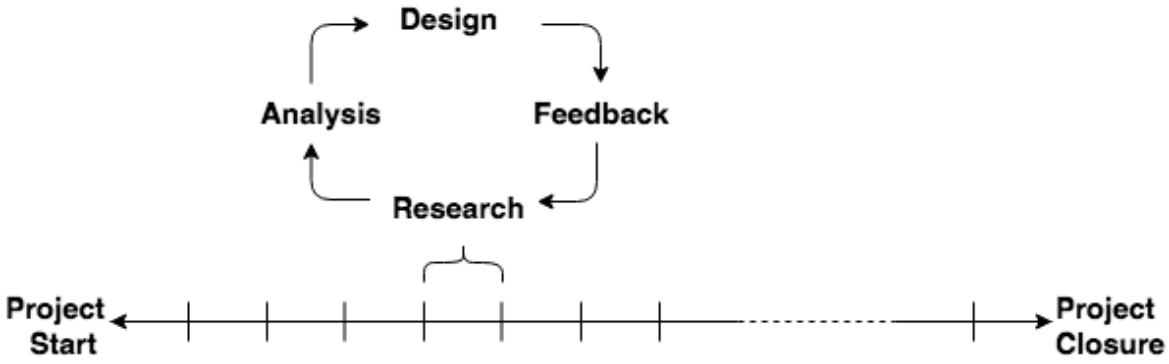

Figure 1.4: Project methodology

The various cycles throughout the project is based on four constituent phases; Research, Analysis, Design and Feedback. First, research is carried out to provide a general overview, which can then be analyzed to determine the project design. Lastly, this is submitted for feedback, which is employed together with the next cycle. These cycles are planned in two week intervals, although the length may vary depending on appointed tasks and schedules. Here, it should be noted that this methodological approach runs in parallel to the development methodology described in chapter 5.

The research methodology used throughout this thesis is abductive in nature, as it starts from the point of view of currently existing and developing technologies and specifications within the problem space. The aim is trying to find a solution to challenges associated with these, as specified in section 1.2. It is not the intention of this thesis to find a be-all-end-all solution to the challenges associated with SDN implementations, such as discussed in the limitations set in section 1.5 and 6.2.

Chapter 2

State of the Art

As delineated in the previous chapter, the problem space of the thesis encompasses the incorporation of virtualization techniques and technologies that can be employed to develop the proof of concept prototype. This leads into the overall objective on how a emulated testbed may be developed to test and monitor SDN controller and OpenFlow interactions over underlying emulated network topologies. An sub-objective is how this may be employed to facilitate the transition from traditionally monolithic to software-defined infrastructures.

However, in order to achieve that objective, it necessitate that one understand the problem space within which SDN operate and how these and existing technologies apply in solving concurrent challenges faced by CSPs and what can potentially be built upon. The aim of this chapter is therefore to presents technologies and existing solutions related to SDN. It serves as the reference frame for subsequent chapters, by establishing the principles and terminology associated with these.

In that regard, there are three distinct fields that are relevant to the proposed prototype; virtualization and emulation of networking nodes, network technologies and protocols and measurement statistics. In this context, the State-of-the-Art architectures, frameworks and technologies and existing solutions that are relevant to the the expected outcome, are briefly discussed in terms of their operational constituent functions. These are summarily compared and analyzed, in order to develop a corresponding implementation strategy. The implementation strategy and development methodology, is presented in chapter 5 - Implementation.

2.1 Software-Defined Networks

SDN is a network architectural paradigm that enables the network control plane logic to be abstracted from the data forwarding plane functions embedded in network hardware and technologies, in a centralized and programmable software platform. This platform is constituted by set of virtualized software functions and open interfaces i.e. Northbound and

Southbound APIs that can be queried to dynamically manage network behaviour between higher or lower layer network components. This is in direct contrast to black box middleware and proprietary interfaces [37]. It is these APIs, alongside SDN controllers, that will be used throughout the thesis project to provision and expose network services, such as, modifying datapaths and queues, routing metrics and providing protocol and technology extensiblility to the network emulation prototype. However, in order to understand the inner workings of these interfaces and functions, the ONF SDN reference architecture and commercial solutions needs to be investigated.

In relation to commercial architectural solutions, there is an preliminary generation being put forth by vendors, e.g. Cisco, Juniper, Ericsson, Huawei, Oracle and Verizon, supporting technological and standardization bodies, e.g. ETSI, OpenStack, ONF and 3GPP [30], and orchestration platforms, e.g. Kubernetes, OpenShift and Mesos. These market actors are expanding their service suites by innovating and adapting to the disruptive change in legacy value chains. This includes the creation and marketing of SDN architectural platforms and Virtual Network Functions (VNF). However, while these initiatives promote the adoption and integration rate of SDN, there are still issues concerning incompatibility between proprietary vendor VNFs and general standardization maturity [35][38][26]. This makes it difficult to frame an architecture that can implement these. A logical approach, as suggested by Ericsson, is to employ a common platform that employs open source initiatives, such as OpenStack and OpenDaylight to enable the reuse of application components [16]. OpenStack can be observed to be implemented many architectures, including Motorola - refer to interview.

These vendors offer a wider selection of competing commercial architectures and platforms and these take on a variety of forms, but at its most simple, the SDN architectural paradigm centralizes control of the network intelligence and state by abstracting applications and control logic from the underlying network elements and resources. All SDN solutions have some version of an SDN controller. Each of these controllers typically support three functional interface types, Northbound, Southbound and OSS/BSS management interfaces, where some may additionally have East- and West-bound interfaces for logically distributed architectures.

These SDN solutions are often determined at the operating Data-Controller Plane Interface (D-CPI), e.g. OpenFlow Protocol, and Application-Controller Plane interface (A-CPI), e.g. SDN applications and management interface, as well as the employed controllers, applications and plug-ins - A-CPI and D-CPI are colloquially called North- and Southbound APIs / interfaces (NBI/SBI) in research. Furthermore, these commercial architectures also incorporate varying degrees of NFV interoperability [39]. In general, it can be argued that there is no single best solution or combination of SDN components that suits all use cases, as each of them have immediate benefits and trade-offs. Thus, each respective enterprises have to decide on a particular SDN solution or a combination of components for implementing SDN in their corresponding networks [26][25][37].

As briefly discussed, the SDN reference architecture may encompass multiple different

network technologies and resources [37][21]. These can be utilized by internal or external SDN applications to perform tasks to expand upon network functions in a SDN environment, allowing operators to respond quickly with flexible network state changes, in order to meet continuously evolving network requirements [40][16]. These applications may concern network monitoring and intelligence, bandwidth management, distributed control and cloud integration and automation, orchestration, and more [16]. These may have a profound impact on existing Operations Support Systems (OSS) and Business Support Systems (BSS) solutions [41], whether these are situated in telecommunications, private or commercial clouds [16]. Orchestration platforms and open source initiatives, such as the one previously mentioned, may assist in facilitating these in the cloud.

In Figure 2.1, a logical view of the base SDN reference architecture, which is proposed by the ONF [37], is presented. This constitutes the frame for the subsequent discussion of SDN components.

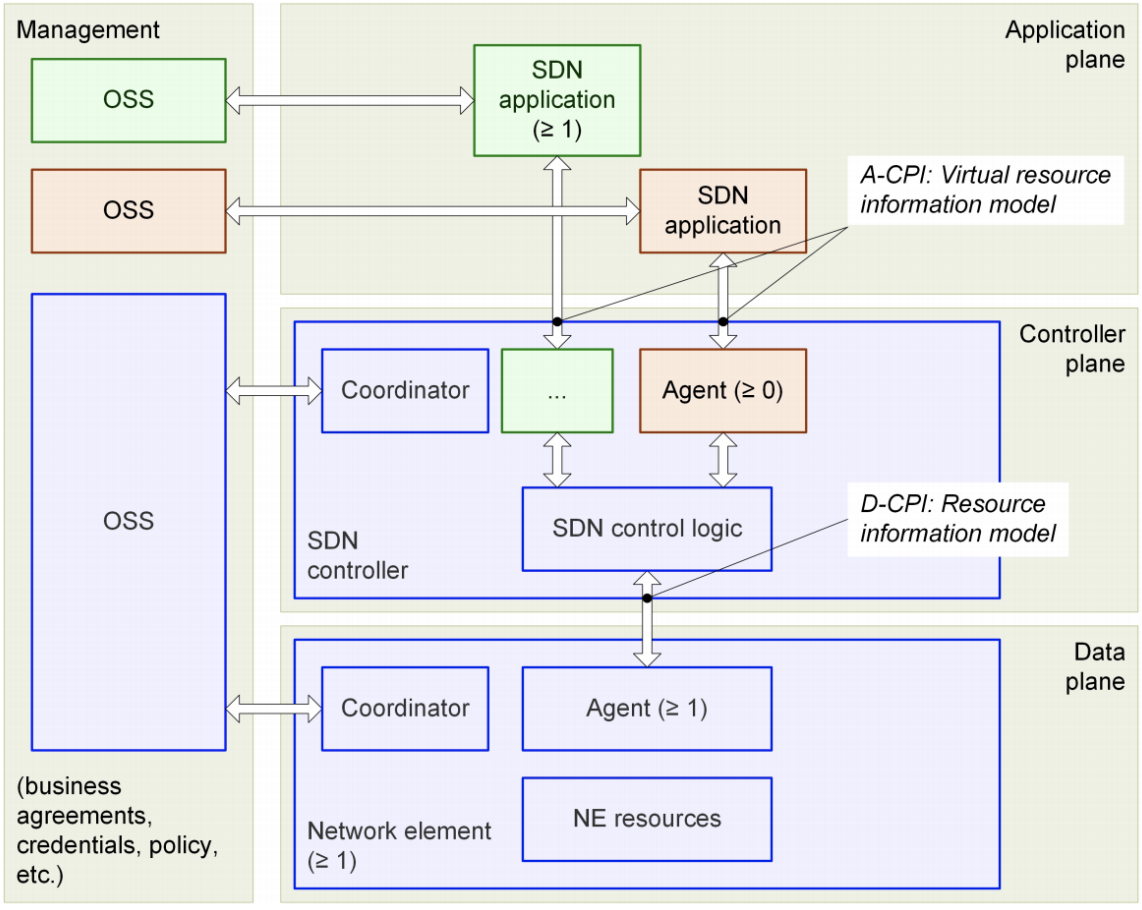

Figure 2.1: A logical view of the basic SDN architecture and components [37]

SDN principles of exposing abstract network resources and state to external applications, permit SDN applications in the application plane or layer, to specify the resources and behavior they require from the network, within the context of business and policy agreements that has been stipulated. The NPIs connects the SDN controller to the application plane and vice versa. An SDN application may itself support a NPI agent, which allows for recursive application actions across functions and policies. Different levels of application hierarchies depends on the degree of abstraction being pursued. Component application may be structured into a service chain, e.g. Identity and Access Management - each level may exist in a different trust domain [37]. Any SDN application(s) communicate its requirements across the NBI, i.e. it may query external services or call SDN controllers across multiple NPI interfaces to achieve its immediate functional objectives and operational requirements. For the management interface, the supporting OSS/BSS and coordinator functions may expose configurations, functions and policies, as these applications require information about their operational state and roles [37]. Any Activity across the NPIs involves queries to the state of the virtual network and invocations to alter said state, e.g. creating or modifying network connection configurations or policy functions between data plane interfaces. Here bandwidth and QoS/QoE parameters may be specified. The NPI may also be used for implementing additional feature, e.g. may function as an access point to configure a service chain (hierarchy) through levelled services or as an input to control virtualized network functions [37].

The SDN controller plane may be constituted by more than one SDN controller, dividing up the logical control plane. The SDN architecture does not specify the internal design of an SDN controller and its logic, as such it can be viewed as a black box, but it is generally agreed that SDN control is logically centralized (may be distributed), working within a scope of subnets, spans physical or virtualized Network Elements (NE) and has no resource contention with other SDN controllers [37], i.e. it can be said to be defined by it externally observable behaviour, e.g. topology knowledge or path computation.

The SDN architectures does however conceptualize a minimum set of functional components within the SDN controller. The Data Plane Control Function (DPCF) component delegates the underlying NE resources currently accessible, per the instructions that has been received from the respective OSS/BSS, coordinator or virtualizer that controls said resources. These are represented in a so-called information model instance that is accessed through disposed agents in a lower level. A Resource DataBase (RDB) concurrently models the information model instance and the necessary supporting capabilities that are available. The master RDB is a repository of all resource information known to the NEs in the accessible subnet, whether physical or virtual. This is especially needed as the scope of an SDN controller is expected to span multiple NEs - unique SPI instance tied to each. Lastly, the DPCF typically includes a function that operates on the aggregate NEs. This function is commonly called application orchestration [37].

- The Coordinator is a component of any given SDN controller that acts in liason with the manager. In order to effectively set up both client and server environments

and be aware and able to change network parameters, there is a need to implement appropriate management functionality across data, control and application plane models, so that information is available [37].

- The Virtualizer is an entity that supports the information model instance at a given NPI. It outlines the trust domain boundary between the different level and the associated application agents. This agent represents the application view of the information model instance. A virtualizer is instantiated by the OSS/BSS or coordinator for each of the application in play [37]. Any number of NBI instances may be concurrently supported by one or more SDN controllers. This is typically carried out when multiple applications are being serviced by one of the SDN controllers.

- The agent in any given SDN controller and at any given hierarchical level, represents the resources and queryable actions available to a given SDN application. A controller-agent model described the relation and status between NEs and SDN controllers. This can be summarily noted from the previously mentioned recursive application hierarchy roles [37].

As described, the control plane is abstracted from the data forwarding plane. The data plane is tasked with implementing routing and switching decisions concluded in the above controller plane. In principle, this plane does not make autonomous forwarding decisions and would need to be directly configured to take any autonomous actions. An example thereof is that the controller plane may configure the data plane to respond autonomously to certain events tied to certain applications [37]. This may be changing to a fallback data forwarding route if one of the the primary logical links fails in a network slice.

The SBI interface between data and controller planes provides feature such as, programmable control of the functions exposed by the RDB and NE capabilities advertisement and event notification. The data plane agent is the entity that executes SDN controller instructions in the data plane. The data plane coordinator is the entity by which management allocates data plane resources to various client agents and establishes policies to govern their use. Agents and coordinators serve the same purpose in every plane of the architecture [37]. Any number of SBI instances may be supported by an SDN controller that is associated with any number of NEs.

2.1.1 Openflow protocol and switch specifications

As delineated in the previous section, the SDN Controller is the primary logical operating entity in most given SDN-based network. This is in charge of relaying information to and from underlying physical or virtualized NEs. As further discussed, standardization organizations and specification bodies have been promoting the federation of SDN architectures and controllers. This entails the use of standard application interfaces. This has lead to the OpenFlow protocol and Open Virtual Switch DataBase (OVSDB) management protocol being de-facto standards for many SDN network operations. As such, any node

that wants to interconnect with an SDN-enabled network, i.e. the SDN controller through the SPI, must support the OpenFlow protocol [20]. As such OpenFlow compliant switches come in two variants, OpenFlow-only and OpenFlow-hybrid. The latter supports additional switching protocols, e.g. MPLS, and can be argued to be a critical component in hybrid networks.

The OpenFlow protocol, or OpenFlow Switch Protocol, is a communications protocol that gives a SDN controller access to the datapath of a set of given NE within a network through a Openflow channel interface, connecting a OpenFlow-enabled switch to an OpenFlow (SDN) controller via a connection URI. This enables a logically seperated administration of switch flow tables through a series of message commands, e.g. by adding, modifying or removing packet matching rules and actions in response to network events, with a one-to-one or many-to-many channel relation between switch and controller. The interface is implementation-specific, but needs to be formatted in accordance with RFC 3986 (URI syntax) [42] and the OpenFlow specification [20].

The OpenFlow protocol is implemented using OpenFlow messages transmitted over the OpenFlow channel Each message type is described by a common message structure, detailing port, header types, flow match, stats and instruction and action. Dependent specific messages such as Switch-to-Controller, asynchronous and symmetric messages, this structure may vary. It does however always start with the OpenFlow header of a given message type structure [20].

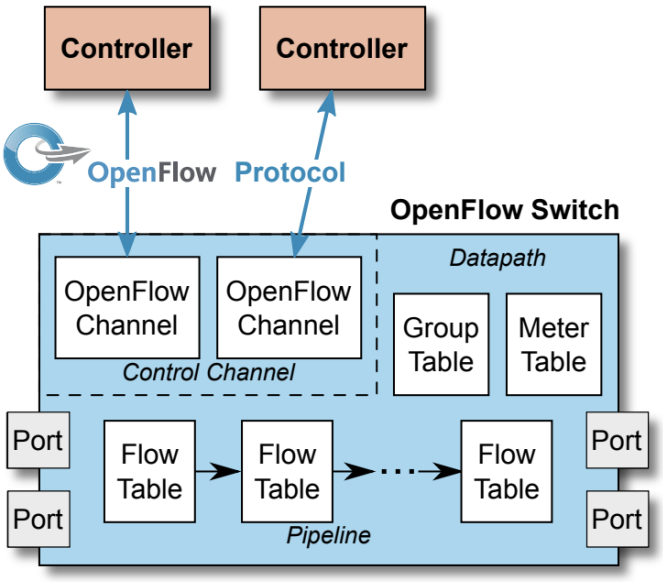

Figure 2.2: OpenFlow-enabled switch architecture and its constituent components [20]

As can be seen in Figure 2.2, an OpenFlow-enabled switch consists of three primary components. Refering to section 2.2.1 and 2.2.2, a datapath consists of a pipeline of flow

tables, a group table, and a meter table and ports, which presents switches with where and how packets are to be received, forwarded and metered based on their respective table entries. A TLS/SSL enabled control channel between the switch and its associated controllers, where command can add, update and delete table entries to meet changing network conditions. The OpenFlow protocol itself, which defines and a standard approach for communication between the switch and external elements [20].

As described above, the pipeline processing, whether at ingress or egress, of packet involves a series of flow tables, each with their respective entries that can is used to perform packet look-ups, forwarding and modifications. Packets traverse the flow tables starting at flow table 0. Figure 2.3 presents the primary component of flow entries in the pipeline of flow tables within a OpenFlow logical switch. Figure 2.3 and the subsequent figures 2.4 and 2.5 presents the primary elements, which are important to understand the inner workings of the Open vSwitch and the Basic OpenFlow User-space Software Switch and employ these as part of the proposed prototype, e.g. feedback statistics. These will be elaborated upon in the subsequent sections.

Figure 2.3: The main components of a flow entry in a flow table of an OpenFlow switch [20].

As seen in figure 2.3, each flow entry in the flow table consist of a series of instruction fields that prescribe how packets should be handled. These are uniquely identified by the match and priority fields. The Match Fields can match incoming packet header fields, including specified ingress port(s) and general metadata values in other processed pipeline flow entry fields with incoming packets. The Priority field describes the order in which incoming packets are matched to the flow entries in the flow table, in descending order. The Counter is one of the main elements in the OpenFlow specifications statistical capabilities, typically counting packets and bytes at specified port or flow entry check points. Instructions or instruction sets are attacked to a flow entry to prescribe how processing should be carried out for incoming packets, i.e. it either modifies or adds actions to the lists of actions being concurrently applied, thus altering pipeline processing. Timeouts define the allowed maximum or idle time before a specific flow entry expires. Cookies is used by the external controller for flow entry filtering through flow requests, where Flags alter the way flow entries are managed [20].

The OpenFlow pipeline processing further involves a group table with its associated group entries. Flow entries in flow tables can point to these during pipeline process to prescribe additional methods of forwarding, e.g. flooding, multi-path and ling aggregation. Each group entry in a group table consists of four instruction fields - the counter has been omitted. The group identifier uniquely identifies a group via a 32-bit value and the group type determines the syntax. The action bucket is a set of actions that are aggregate applied, e.g. modifying a packet and forwarding it to an egress port. These may be chained.

In figure 2.4, the main components of a group entry in the group table of an OpenFlow switch [20].

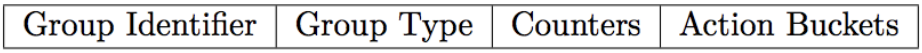

Figure 2.4: The main components of a group entry in the group table of an OpenFlow switch [20].

Furhermore, the OpenFlow pipeline processing involves a meter table with its associated meter entries. Flow entries in flow table can point to these during pipeline processing to trigger QoS policing operations on ingress packets. Each meter entry in a meter table consists three instruction fields - counters are omitted. The meter identifier is a 32-bit unique integer identifying the meter. The meter band is an aggregate list of meters, where each meter band specifies the rate of packets and how these should be processed when attached to a flow entry. Each meter band is further identified by its band type, rate, counters, and type specific arguments. In figure 2.5, the main components of a meter entry in the meter table of an OpenFlow switch is shown [20].

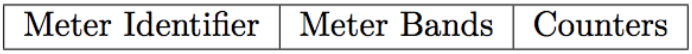

Figure 2.5: The main components of a meter entry in the meter table of an OpenFlow switch - Proposed in the OpenFlow switch specification [20].

The OpenFlow pipeline processing can be further illustrated by a flow diagram. This figure provides a describing over the operation logic, which is the different actions, sets of actions and action buckets that are being applied to a packet and how the different tables, entries and counters relate to each other, providing an understanding of the inner workings of OpenFlow-enabled switches as a packet traverses it. In Figure 2.6, a flow diagram of the OpenFlow pipeline processing [20] is presented. Here is should be noted that OpenFlow v1.4 introduced bundles, which is a sequence of modification requests from the controller that is applied as a single operation. Furthermore, that all processing was done in the context of the input port up until OpenFlow version v1.5, which introduces Egress Tables, enabling processing to be done in the context of the output port [20].

The OpenFlow specification does not stipulate data structures and algorithms to implement the pipeline of the switches that support the protocol. As long as the implementation follows the described behavior, there is freedom to define the structure of components. An example of this is OVS employing the OVSDB management protocol instead of OF-Config. These capabilities are also being implemented in many of the network emulators described in section 2.7.

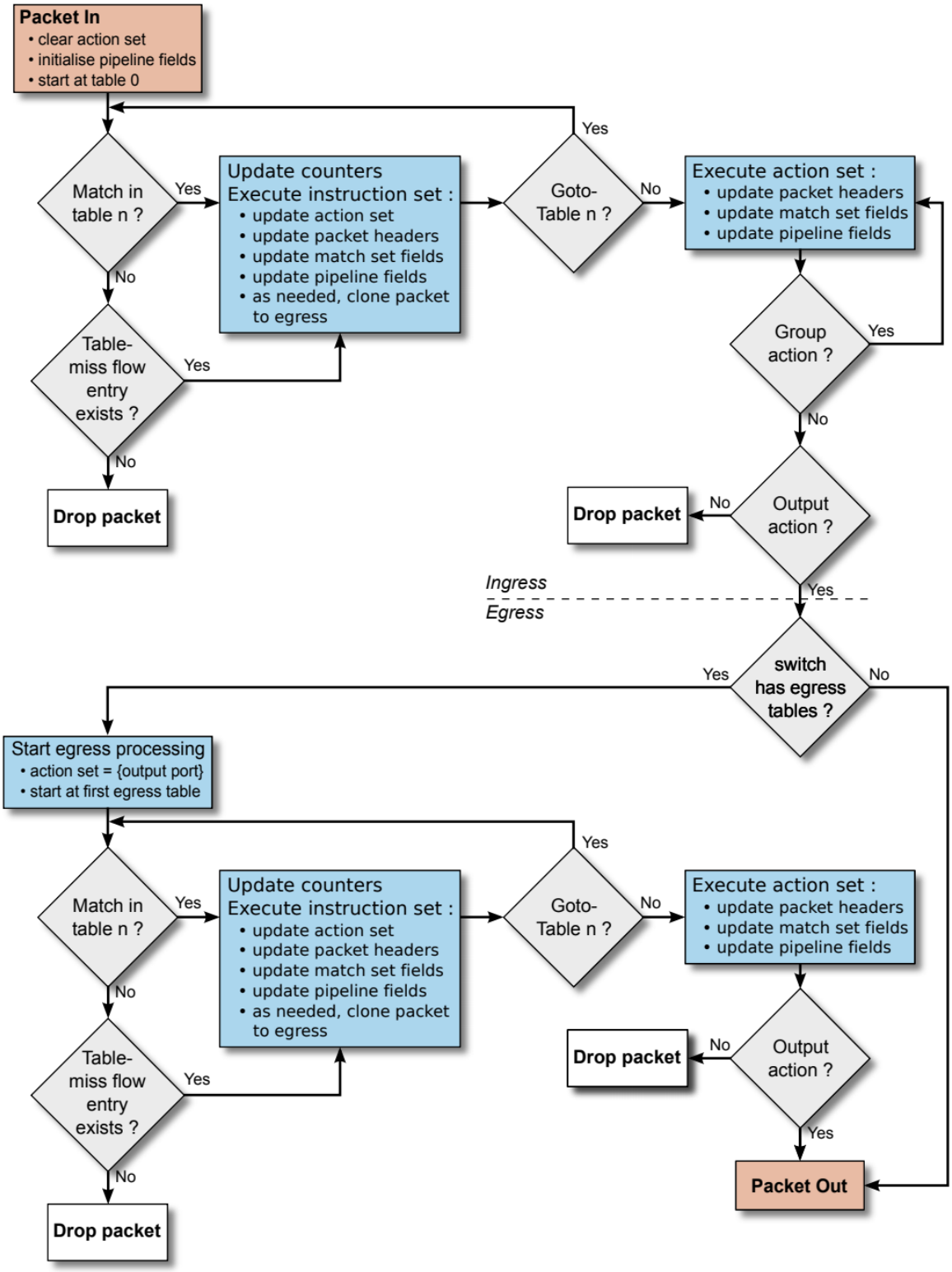

Figure 2.6: A flow diagram of the OpenFlow pipeline processing [20].

2.2 Virtual Switches

As the prototype is concerned with emulating network infrastructure topologies at the data layer in the SDN reference architecture, with a focus on switching fabrics. It is then necessary to investigate the functionality of openflow-capable virtual switches.

2.2.1 Open vSwitch Specification

The Open vSwitch (OVS) is an open source implementation of a virtual multi-layer switch, which is compatible as a software switching node for linux-based virtualized environments and as a control stack across physical servers, providing flexibility for virtual and physical platform implementations. It is specifically designed to enable large scale network automation, through remotely programmable software extensions and control of forwarding functions. Network integration is achieved by exposing its standard control and visibility interfaces to external platforms, service and technologies [43][44], e.g. OpenFlow-based SDN controllers and OVSDB management protocol[45]. This allows for OpenFlow to be embedded in networks and achieve the benefits of SDN. OVS is meant to be controlled and managed by third party controllers and orchestration solutions. Furthermore, it supports industry standard configuration and management interfaces and protocols, e.g. NetFlow/IPFIX, SNMP, SPAN, port mirroring and CLI [43]. These will be presented in greater detail in subsequent chapters.

The OVS is supported by multiple hypervisors and Linux-based kernel and user-space solutions, both commercial and open source, almost becoming the de-facto virtual switch for the majority of network emulators for research and development, e.g. the Mininet network emulator implements the kernel space OVS for prototyping of SDN architectures [46]. In Figure 2.7, a high-level overview of the OVS architecture is shown.

As described and can be seen in Figure 2.9, the OVS can be ported to both software and hardware platforms, i.e. any type of data plane (NIC). As illustrated above the OVS architecture consists of a series of components that enables the aforementioned functionality. The ovs-vswitchd is the daemon, i.e. user-space program, which implements the OVS switch alongside companion Linux kernel modules (libraries) that enables flow-based switching. This daemon can query a network state database, the ovsdb-server, over an Inter-Process Communication (IPC) channel to obtain a desired OVS configuration, which it passes to the ofproto library. It may further pass statistics back to the ovsdb-server. The ovsdb-server consists of configuration details concerning bridge, interface, tunnel configurations and OpenFlow controller URI [43][45]. It can be remotely configured by an OVSDB manager via the OVSDB management protocol - elaborated upon in section 2.6.2.

OVS and OVSDB provides a series of base commands (script extensible) that can be used for querying, updating and deleting configurations, OpenFlow-enabled switches, datapaths and OVS daemons. The most commonly applied are vsctl, ofctl, dpctl and appctl [43][20].

The ofproto library implements OpenFlow interfaces and communicates with OpenFlow controllers and external software through an ofproto provider. For virtual implementations, the ofproto provider (dpif provider) consists of the ofproto-dpif library that is used for manipulating datapaths or dpif (flow table entries). On packet ingress the dpif library performs a look up and perform designated actions - this functionality is often delegated to the ofproto-dpif. This task may also be passed to a OpenFlow controller - refer to figure 2.6 for reference [43].

The netdev library is a module that operates over the netdev provider, it abstracts ethernet interfaces. Where the netdev provider implements the actual interface, specific to the Operating System (OS). Both of these are critical to fulfilling OpenFlow functional requirements [43].

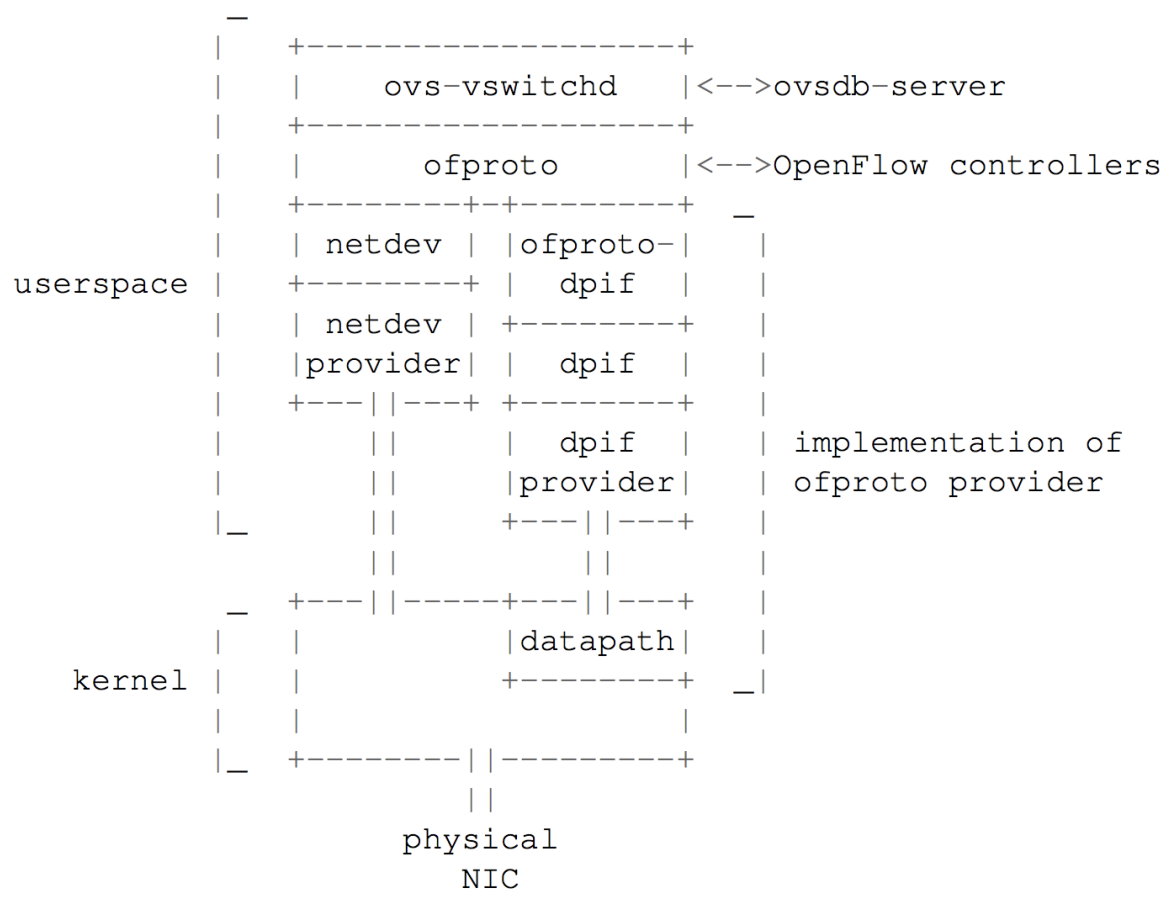

Figure 2.7: The high-level architecture of Open vSwitch [44][20].

2.2.2 Basic OpenFlow User-space Software Switch specification

BOFUSS is an open source programmable virtual switch, developed with the main objective of maintaining simplicity across its core functions. As such it has a limited suite of features compared to OVS, but remain relevant for learning purposes. The design and implementation of components are limited to the OpenFlow protocol and switch specification. As many of the functional features of BOFUSS, i.e. Openflow, has been described in section 2.1.1 and 2.2, components will be referenced by name, but will not be elaborated upon further [47]. In Figure 2.8, the high-level architecture of BOFUSS is shown.

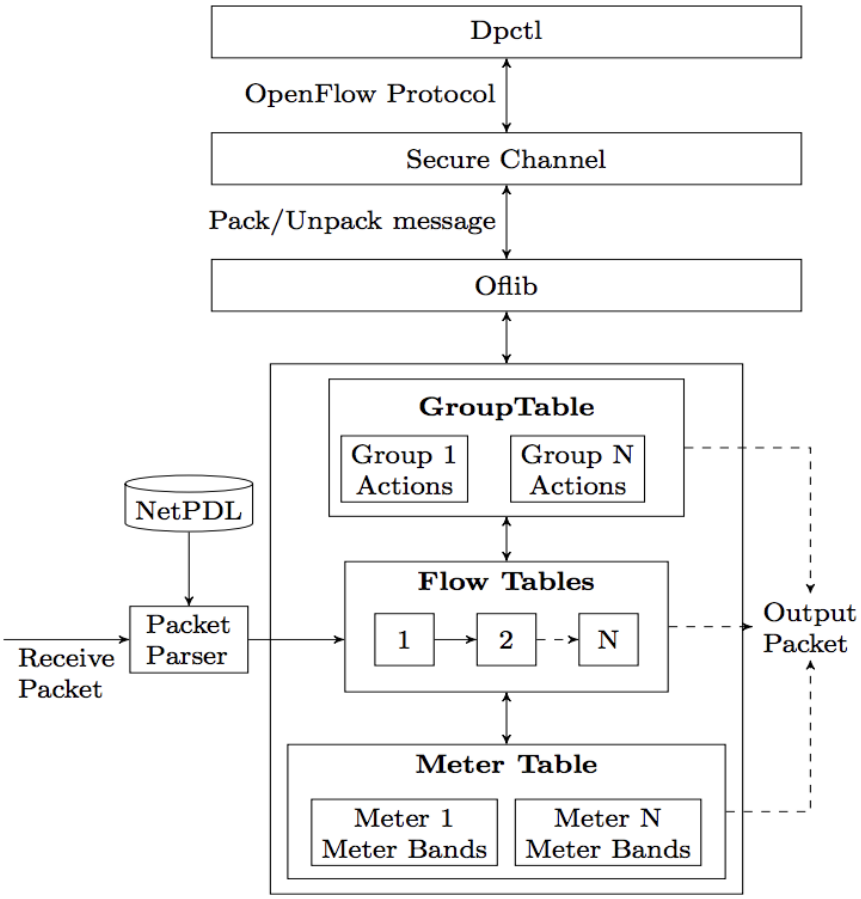

Figure 2.8: The high-level architecture of BOFUSS [47]

The Oflib module library converts OpenFlow messages to a BOFUSSS network data structure format (packing) and conversely when relaying statistics (unpacking). These are commonly bit-wise operations to avoid system resource hogging, but may be more complex depending on the message and associated functions. Ingress pipeline packet has

its header field check by the packet parser component, akin to the OVS ofproto-dpif, where they are matched against flow entries in a flow table with its associated instructive actions. The group table enables different action buckets to be set up to handle specified events, e.g. port fail-over and link aggregation. Meter table connected to flow tables can perform per flow QoS-related operations, e.g. rate-limiting [47].

The Secure Channel is a external function that handles TLS/SSL setup between the virtual BOFUSS switch and the SDN (OpenFlow) controller URI. This is dissimilar to how it is handled in OVS, as the OpenFlow protocol and switch specification does not specify a connection method, i.e. protocol. Lastly, the BOFUSS switch presents a CLI tool, Dptcl that enables simple configuration and monitoring commands to add, update or delete table entries, in order to check and modify the current state of the switch [47].

2.3 SDN controllers

As the potential candidates for the data layer switching fabric has been established, it is then logical to investigate the control layer and which SDN controller candidates there are and what functionality they provide. Because these controllers have to integrate with an emulated environment, it is a requirement that these are open sourced and supports the OpenFlow protocol. Multiple market actors have developed and implemented these controllers, either as standalone or as part of orchestration platforms.

- **OpenDaylight (ODL)** - The OpenDaylight project is an open source platform that uses open protocols to provide centralized, programmatic control and network device monitoring. It has a micro-services application architecture (Apache Karaf OSGI components) with support for a wide and growing range of network configuration protocols, e.g. NETCONF, OVSDB management protocol and SNMP [48].

- **Floodlight** - Is a OpenFlow controller. It realizes a set of common functionality to control a OpenFlow-capable network, while packaged applications on top of it realize different features. The applications are built as Java modules compiled with Floodlight the REST API. This enables extraction of topology views, routing metrics, bandwidth statistics, etc. [49].

- **Nox** - Nox Classic and NOX is the original OpenFlow controller. It serves as a network control platform core that supports control methods, such as network packet process and process threading and OpenFlow event engine, in addition to OpenFlow APIs [50].

- **Pox** - POX is a Python-based open source OpenFlow Controller variant of the NOX controller. It is intended to be used as a learning tool. The POX framework contains a series of stock components, core APIs and event handlers that allows for quick deployment [51]. It and Nox comes pre-installed in many SDN-capable network emulators.

- **Beacon** - Beacon is a Java-based OpenFlow controller, which has served as the foundation for the Floodlight controller. It was built on an OSGI framework and leverages open source libraries. It has multiple reference applications that builds upon its core platform, adding additional device management, topology view, link and routing computation and web-UI APIs [52].
- **Ryu** - Ryu is a component-based SDN framework. Ryu provides software components and REST/RPC APIs that allows for quick development of network management and control SDN applications. These components include link aggregation and traffic monitoring APIs. Ryu supports various network configuration protocols, e.g. Netconf and OpenFlow-config [53]
- **ONOS** The ONOS controller is implemented as a collection of OSGi Java applications (Apache Karaf components) that interact through Java APIs and REST APIs. ONOS provides instruments for developing new SDN applications, including templates for integration into its CLI and GUI. It provides two main northbound interfaces: the global network topology view and the intent (policy) framework [54], .

	Programming Language	GUI	Docs	Modularity	Architecture Support	Northbound APIs	Southbound APIs	OpenStack Support
ONOS	Java	Web-based	Good	High	Distributed	REST	OpenFlow 1.4, NETCONF, SNMP	Yes
Open-Daylight	Java	Web-based	Good	High	Distributed	REST	OpenFlow 1.4, NETCONF, SNMP, OVSDB	Yes
Nox	C++	Python / third party software	Poor	Poor	Centralized	REST	OpenFlow 1.0	No
Pox	Python	Python / third party software	Decent	Poor	Centralized	REST	OpenFlow 1.3	No
RYU	Python	Python	Decent	Decent	Centralized	REST	OpenFlow 1.4	Yes
Beacon	Java	Web-based	Decent	Decent	Centralized	REST	OpenFlow 1.3	Yes
Floodlight	Java	Web-based / Java	Good	Decent	Centralized	REST	OpenFlow 1.4	Yes

Figure 2.9: Table detailing a comparison of open source SDN controllers

Figure 2.9 presents an overview over the different features that are natively supported by the delineated open source SDNcontrollers. Here it should be noted that functionality

and capability of these can be extended extensively through the employment of third party modules and solutions. These open source controllers contain a collection of open modules and interfaces that can be manipulated and queried to perform different actions in respect to integrating them with the network emulators described in section 2.7.

2.4 Network Function Virtualization

As the thesis concerns itself with employing virtualization technologies and emulated environments, and having in so far explored SDN, OpenFlow and virtual switch architectural components, there is merit in investigating how these can be employed together with NFV, or if the logical approach can be implemented similarly in an emulated environment. To do so, the Network Function Virtualization (NFV) architecture will be illustrated in figure 2.11 and described shortly.

An example of how these components can be employed, is among the Network Function Virtualization Infrastructure (NFVI), as this is a common approach for providing connectivity and packet steering among Virtualized Network Functions (VNF), substituting Management, Automation and Network Orchestration elements to accelerate deployment [39].

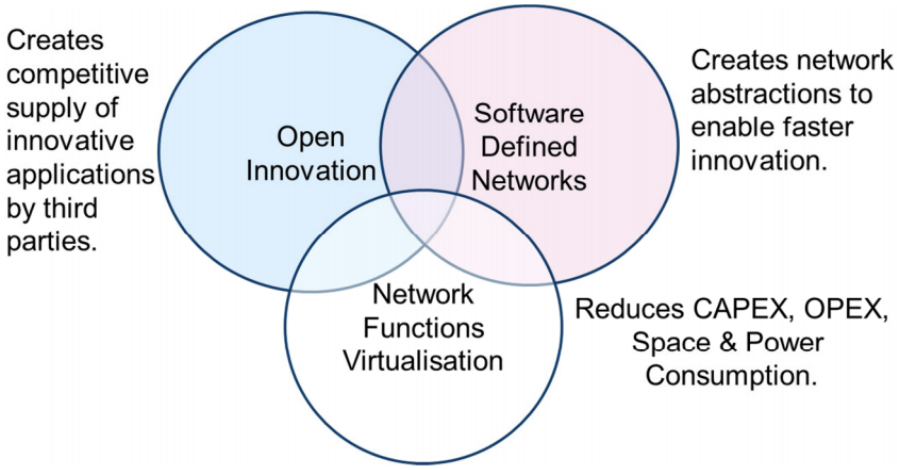

Figure 2.10: The relationship between NFV and SDN [55]

SDN and NFV are highly complementary technologies, but both architectures are independent of each other and can be implemented as standalone service architectures. However, there is a myriad of commercial and open source implementations across both technologies, as the interconnection between these provide compelling opportunities for CSP networks, e.g. SDN enables policy-based network forwarding automation and provisioning, where NFV ensures network service availability across a virtualization layer working on infrastructure and orchestration platforms [56]. Thus providing increased network

resource flexibility and efficiency across lower and higher layers, while also counteracting vendor lock [39]. As such, a logically combined integration of both concepts allows for open innovation across CSP networks and can enable eased integration of the respective technologies [55].

The NFV architecture is presented in figure 2.11. It is consists of three main functional blocks and its associated interconnected interfaces, as detailed below.

- Virtual Network Functions (NFV) - A specific virtualized network function, e.g. router, firewall or load balancer, running within a Virtual Machine (VM) or container. These may be constituted by one or more VNF Components (VNFC), which is packaged functions enabling a implementation specific VNF. These components are connected to the NFVI block through the external Vn-nf execution environment reference point, which is the VM container logical interface, through which instantiating of VNFs occur. The VNFs and EMs are connected to the VNF Manager (VNFM) through the Ve-vnfm interface [57].

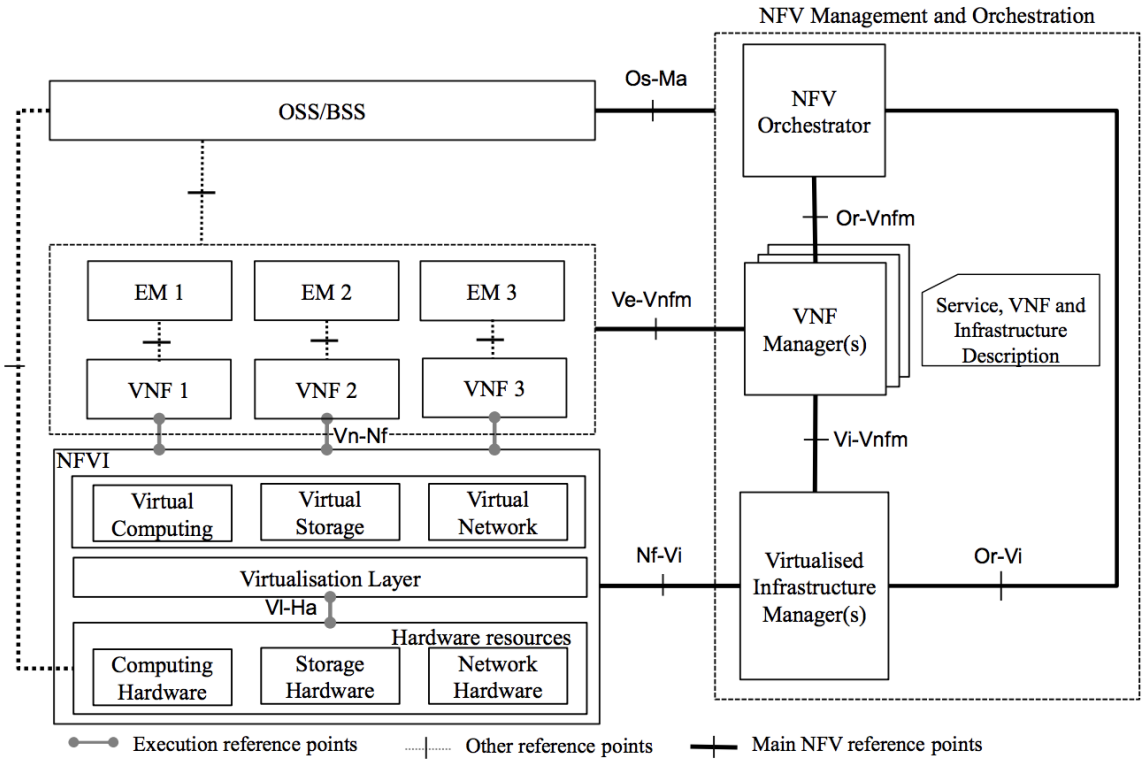

Figure 2.11: The Network Functions Virtualisation architectural framework [58]

- NFV Infrastructure (NFVI) - Comprises the hypervisor, compute and networking infrastructure domains required to deploy, operate and monitor VNFs, within which

the virtualization layer abstracts software functions from underlying hardware resources through the internal Vi-Ha execution environment reference point. This layer may for example be composed by OpenFlow and OVS. It is connected to the NFV Management and Orchestration block through the external management and orchestration reference point between agents in the respective domains and the Virtual Infrastructure Manager (VIM) functions between which requests for hypervisor services occur [57].

- NFV Management and Orchestration (MANO) - Is comprised of the VIM which controls the interaction between VNFs and underlying physical resources, i.e.. allocation. The VNFM which handles the instantiating of NFVs, i.e. initialization and termination. The NFV Orchestrator (NFVO) which handles registration of network services through the Or-VI interface and subsequent VIM functionality, i.e. operations and performance management [57][59].

As mentioned through the past sections, OpenStack is commonly employed by orchestration systems presented market actors. Here it is often employed in conjunction with NFV in data centers to deploy cloud services, by enabling CSPs to deploy VNFs using COTS hardware. Notably, both the ETSI OpenSource Mano (OSM) [60] and the Linux Foundation Open Platform for NFV (OPNFV) [61] have used OpenStack as their VIM [16].

In theory it should be possible to connect a given network emulator, if the emulated network can be exposed to the internet, to the OpenStack service to through Openstack Quantum and Openstack Nova Compute APIs. The latter provides an approach for provisioning virtual computational environments and instances, i.e. containers and virtual machines. However, the support for containers is limited [14].

2.5 Internet protocol Suite and networks

As most integration of SDN- and NFV-enabled architectures will either be hybrid solutions or require a prolonged migration period in which existing technologies are supported [25][31], there is an inherent need to support the implementation and employment of protocols that can be utilized together with or instead the OpenFlow protocol for the management of non-OpenFlow-enabled network forwarding functions and mechanics. In the case of the thesis, this pertains to switches within the data center switching fabric. The integration of these with SDN controllers may be achieved through Westbound, Eastbound or the Southbound interface. Many protocols and mechanisms, such as MultiProtocol Label Switching (MPLS), Border Gateway Protocol (BGP), SNMPv3 and NETCONF/IPFIX, are concurrently being used together with SDN and NFV technologies in existing networks and hybrid solutions [17].

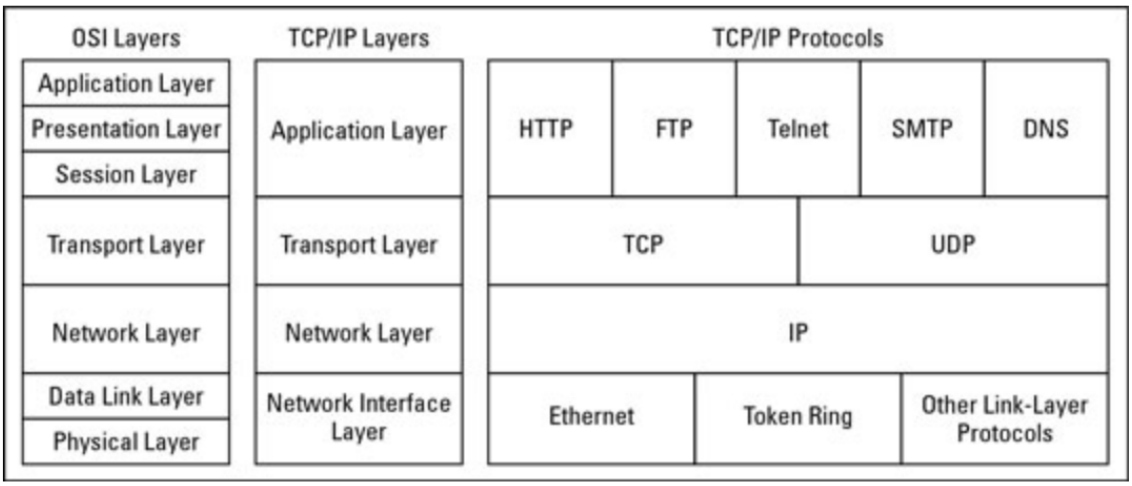

Figure 2.12: The basic architecture of TCP/IP (Internet protocol suite) [62]

The TCP/IP (Transmission Control Protocol/Internet Protocol) is the most commonly used model used to represent the IP suite and will be referred to consecutively throughout the report. The IP suite is a set of communications protocols that define the networking model for inter-connected networks, of which several SDN and NFV components build upon. It follows a layered architecture involving the grouping of protocols into layers based on their generic functionality - the layers are abstracted through encapsulation. It consists of four layers of abstraction [63][64], which corresponds with the 7 abstraction layers defined in the OSI model. In Figure 2.11, the basic model of the TCP/IP and its correspondence with the OSI model, and a subset protocols is shown.

2.6 Network Management Technologies

Any of the chosen SDN Controller will support OpenFlow for the Southbound interface. However, many controllers may support network management protocols, i.e. network configuration protocols for administrative configuration of the management plane, or network monitoring protocols for relaying network performance and statistics - The available protocol extensions depend on the respective SDN controller. These may extend or support the information available for subsequent aggregation and analysis. In this context, the State-of-the-Art network management technologies that are applicable in SDN Controller Southbound APIs and to the proposed prototype are briefly discussed and compared. As with SDN and NFV, the proposed prototype aims to promote an open architecture, prompting vendor-neutral remote network configuration and monitoring technologies to become increasingly relevant.

2.6.1 Network Monitoring

Network monitoring technologies can be classified based on the level of granularity and features provided in terms of network traffic monitoring and analysis.

Simple Network Management Protocol (SNMP) version 3

SNMPv3 is a network management protocol used to configure and monitor network element functions. SNMP provides a neutral connection method for network elements to relay management data within multi-vendor environments. Through simple get and set command functions between network devices and the management system, it can report on concurrent network metrics. SNMP functions follows a synchronous communication model. There are three main components in an SNMP-enabled network [65][66]:

- **SNMP Agent** - Function runs on the equipment being monitored, collecting data metrics. When queried by a SNMP manager, the agent sends said metrics back to the NMS. An agent may proactively notify the NMS if an error event occurs. Most devices come with an SNMP agent pre-installed [65][66].
- **SNMP manager / Network Management System (NMS)** - Central logical console. It queries agents for updates at regular intervals. Data processing capabilities depends on the NMS [65][66].
- **SNMP Management Information Base (MIB)** - This database is a text file (.mib) describing all objects used by a device that can be controlled using SNMP. Must be loaded into the NMS so that it can monitor the status of these properties. Each MIB item is assigned an object identifier (OID) [65][66].

Internet Protocol Flow Information Export (IPFIX)

IPFIX is a network monitoring protocol used for defining how flow data should be formatted and transported. Flow data is sets of IP packets passing an observation point - inter-operable with OpenFlow flow table and entries. Analysis of said flow data is implementation specific [67][68].

IPFIX defines a general architecture. This entails formatting, i.e message data frame formats, data encoding and information models and a transport protocol based on said format. In short, the architecture specifies three types of processes and associated nodes [67][68]:

- **Metering Processes (MP)** - During packet observation packets are captured from an observation point (flow probe) and formatted. This is typically carried out by a Network Interface Card (NIC) or network node. Within the Metering Process, packets are aggregated into flows [67][68].

- **Exporting Processes (EP)** - Terminated flows, i.e. flow records within a specified interval, are exported by the Exporting Process, i,e, the flow is sent from the observation point to a collection point (flow collector) [67][68].
- **Collecting Processes (CP)** - Entails reception, storage and formatting, e.g. aggregation, filtering and encoding of flow data generated by the previous processes. Analysis may be automated or manually implemented [67][68].

Sampled Flow (sFlow)

sFlow is a packet sampling protocol for monitoring traffic flows. It can be configured using the SNMP management framework [66]. The sFlow monitoring system consists of an sFlow Agent, a central data collector, or sFlow Analyzer and a sflow MIB [69]:

- **sFlow Agent** - Sampling mechanism situated within a network node for monitoring flows. It combines interface counters and flow samples into sFlow datagrams that are sent in a sFlow datagram to an sFlow Analyzer - central logical console[69].
- **sFlow Analyzer** - Centrallized data collection node, i.e. a central server running sFlow function that analyzes and reports on network traffic [69].
- **sFlow Management Information Base (MIB)** - Functions similarly to a SNMP MIB [65]. Defines a control interface for an sFlow Agent. This interface provides a standard mechanism for remotely controlling and configuring an sFlow Agent [69].

Switched Port Analyzer (SPAN) / Port Mirroring

SPAN, sometimes called Port Mirroring, copies flows and forwards it to a SPAN port for analysis by a Network Analyzer, i.e traffic can be monitored on a switch port by forwarding incoming and outgoing traffic to another port for data collection and analysis. SPAN is native to most commercial switches [70].

- **Remote SPAN (RSPAN)**: An extension that allows traffic monitoring from source ports distributed over multiple switches - Enables flow capture centralization. It works by mirroring flows from multiple switch source ports of a RSPAN session onto a dedicated trunked VLAN. On the switch that contains the destination port for the session, traffic from the RSPAN session VLAN is mirrored out the destination port[70].
- **Encapsulated Remote SPAN (ERSPAN)**: Encapsulated Remote SPAN (ERSPAN) employs Generic Routing Encapsulation (GRE) on all captured traffic and allows it to be extended across layer 3 domains[70].

	Sampling Granularity	Monitoring Methods	Transport Protocol	Measuring Applications
SNMPv3	Per interface	Pull-based	User Datagram Protocol	Bandwidth and fault analysis
IPFIX	Per flow	Push-based	Stream Control Transmission Protocol	Traffic analysis
sFlow	Per interface	Push-based	User Datagram Protocol	Bandwidth and traffic analysis
Port Mirroring	Per packet	Mirrored port	Mirrored port interface	Packet inspection and analysis

Figure 2.13: Comparison of network monitoring protocols and technologies

As presented in figure 2.13, each of the described network monitoring protocols and technologies have different approaches and capabilities. There is no single available best solution for network monitoring that fits all use cases. Moreover, there are quite a few proprietary vendor-specific network monitoring solutions that are based on and variants of the stated network monitoring technologies, but none of them are relevant for achieving an open network architecture.

2.6.2 Network Configuration

In regards to network configuration protocols and technologies, these can typically be classified based on the flexibility they provide in relation to configuring networking nodes. Relating to SDN controllers (section 2.3), virtual switches (section 2.2), orchestration platforms (section 2.8.) and network emulators (section 2.7), current practices have the user employing a variety of Command Line Interfaces (CLIs) or web-based GUIs for network configuration. This imposes immediate challenges in terms of cross-integrating. Gathering the aforementioned functionality in one CLI or GUI could be argued to be advantageous.

Given the scope of the problem domain, there are necessary requirements to consider in relation to the implementation of network configuration protocols and technologies. These are that they should be vendor-neutral and open source. Therefore, in extension of OVS, BOFUSS and OpenFlow, the Open vSwitch Database (OVSDB) Management Protocol is presented together with SNMPv3 and NETCONF, as these fulfill the criteria stipulated above - they allow for vendor-neutral and remote configuration of virtual switches through extensible interfaces. An approach that is required to integrate these with the network emulators presented in section 2.7.

Network Configuration (NETCONF) Protocol

The Network Configuration Protocol (NETCONF) defines a simple architecture for network device management and configuration that allows new or existing configuration information to be uploaded and modified. It achieves this through a series of message command using Remote Procedure Calls (RPC). Communication between a network client and server is handled through XML-encoded RPCs over a TLS/SSL connection. NETCONF was designed as the evolution of SNMP and CLI [71]. The protocol that defines configuration datastores and a set of Create, Retrieve, Update, Delete (CRUD) operations that can be used to access these datastores. There are three NETCONF datastores – candidate, running and start-up [71]:

- **Configuration datastore** - Datastore holding the complete set of configuration information required to boot a device from its initial default configuration into a specified operational configuration, allowing for state-wise transition [71].
  - **Candidate configuration datastore** - A datastore that can be manipulated without impacting current configurations across networking nodes. Changes to said datastore can be pushed to the running configuration datastore [71].
  - **Running configuration datastore** - A datastore holding the given configuration currently active on the device. The running configuration datastore is essential to the operational state of NETCONF [71].
  - **Startup configuration datastore** - A datastore holding the boot configuration loaded to network device upon startup. Only required on networking nodes with separate startup configuration requirements to the running configuration datastore [71].

Open vSwitch Database (OSVDB) Protocol

The Open vSwitch Database (OVSDB) is a database server that is accessible by network client nodes through a JSON-encoded RPCs communication protocol [72]. It supports three service models, i.e. standalone, active-backup, and clustered. These service models introduces compromises across consistency, availability, and partition tolerance in database deployments. As such, their use depends on the specific use case [45].

- **Standalone** - Standalone databases run on a single server. If the server operational state is disconnected or data corruption occurs, the database and its contents becomes inaccessible [45].
- **Active-backup** - Active-backup databases run on two parallel servers, with one being the roll-over safety. One is designated in active role state and the other in backup role state. Active servers function similarly to standalone servers [45].

- **Clustered** - Clustered databases runs across three or more parallel servers. Servers dedicated to a cluster automatically synchronize write-over between the allocated servers. The fault tolerance of the cluster depends on the quantity of active dedicated servers [45].

The OVSDB management protocol is employed to ensure state synchronization between networking nodes, as it enables each respective networking node client to monitor the designated database(s). Upon operational state change, i.e. added, deleted or modified entries, the database servers push said changes to the client. This allows networking clients with continuous updates to the operation state of the network [45][72].

The OVSDB management protocol has multiple uses within OVS, with the primary use being configuring and monitoring the OVS daemon mentioned in section 2.2, using OVS configuration and monitoring schemas. OVS includes an OVSDB server that supports other network monitoring protocols through its extensions. It furthermore includes a CLI OVSDB client and OVSDB client libraries [45].

A comparison of between the presented vendor-neutral and open source network configuration protocols and technologies is presented in Figure 2.14.

	Network Configuration Technologies	Managed Network Devices	Managed Network Types	Flexibility	Interoperatibility
SNMPv3	Vendor-neutral	Any device	Heterogeneous networks	Medium	Medium
NETCONF	Vendor-neutral	Any device	Heterogeneous networks	High	High
OVSDB Management Protocol	Vendor-neutral	Open Vswitch	OVS-based networks	High	High
CLI	Vendor-specific	Any device	Homogeneous networks	Low	Low
Web-based	Vendor-specific	Vendor-specific devices	Homogeneous networks	Low	Low

Figure 2.14: Comparison of network configuration protocols and technologies

2.7 Network Emulators

As the prototype seeks to emulate data layer functionality in virtual environments, i.e. containers or virtual machines, in an SDN architecture, in order to test and monitor the in-

teraction and performance of SDN elements with the ultimate goal of aiding technological decision-making. There is an inherent need to investigate open source network emulators and how these may be integrated with the virtual environments, to avoid having to provision real networks for testing. This approach is dissimilar to network simulation, where mathematical traffic models are applied.

Network emulation is a virtualization technique employed for testing performance metrics of network applications over an emulated network, either as part of research developments or as test cases before deploying these to production environments. As such, the operational objectives of such emulators are to observe and analyze performance metrics and the operational effects of changing network architectures and conditions. All these presented objectives are aligned with the requirements constituting the proposed prototype and is situated at the core of the functional architecture.

A multitude of network emulators exists. Each of these have different distinct purposes and performance prioritization in regards to how they are intended to be deployed. Emulators that do not support SDN and integration of Open vSwitch and Openflow reference switches will not be considered. As open source projects, these network emulators provide opportunities integration with different Openflow protocol versions, OVS and SDN controllers. These further allow for programmatic changes of API libraries needed to realize the proposed prototype. Furthermore, these can be run inside virtual machines or containers.

- **Common Open Research Emulator (CORE)** - CORE is a network emulation system for emulating network topologies across multiple VMs or containers. CORE consists of a GUI for configuring network topologies and API library modules for scripting network emulation. CORE can be connected with existing networks [73].

- **IMUNES** - IMUNES is a network emulator framework that can be employed topology configuration and management and presents a GUI application. IMUNES network emulation is facilitated through partitioning of lightweight VMs based on a modified FreeBSD kernel. These VMs can be connected via kernel-level links.

- **Mininet** - Mininet is a network emulation orchestration system. It creates virtual hosts by employing lightweight virtual containers that are built around a Linux kernel. It contains a network namespace mechanism for separating network interface and table [46].

- **NetKit** - NetKit is a lightweight network emulator environment based on User-Mode Linux variant of the Linux kernel employed in most other emulators. Network topologies and configuration files can be created through a GUI or a XML-based (NetML) language.

Figure 2.15, presented on the next page, summarizes and compares the emulators in term of the virtualization libraries and technologies, as well as their presented interfaces,

remote control capabilities, communication overhead and link emulation that each respective emulator provides. These features presents the core considerations related to the proposed prototype and will be discussed in section 3.1.2.

	User Interface	Remote Procedure Calls	Virtualization Library	Virtualization Technology	Host Distribution	Operating System	Node Emulation	Network Emulation	Link Emulation
CORE	GUI, CLI	Binary	API	Container, hypervisor	Yes	Linux, FreeBSD	Jails, LXC, Xen, netns	Linux bridge, Vnet, netgraph	netgraph pipe, ebtables
IMUNES	GUI, CLI	No	Monolithic	Container	No	Linux, FreeBSD	Jails, LXC, netns	OVS, netgraph, Vnet	netgraph pipe,
Mininet	GUI, CLI	No	API	Container	No	Linux	LXC, netns, control groups	OVS, IVS, OpenFlow reference, Linux bridge	Linux TC, Netem
Netkit	GUI, CLI	No	API	Container	Multi-instancing	Linux	UML	UML switch	Linux TC

Figure 2.15: Comparison of open source SDN-capable network emulators

2.8 Virtual environments and orchestration platforms

As mentioned, the prototype seeks to enabled a distributed data layer architecture by deploying and packaging emulated networking nodes inside isolated and encapsulated containers or virtual machines. Here, it should be noted that there is a potential for nested deployments, e.g. containers in a virtual machines.

As presented in figure 2.16, containers and virtual machines presents two different approaches to managing computing or workload environments for the applications hosted on them. Virtual machine software employ hypervisors, which acts as an agent, akin to an SDN agent, providing a partitioning abstraction layer between the virtual machine guest operating system environment and the underlying host operating system and hardware - allowing for different operating systems and bins/libraries across applications on the same node [74]. Alternatively, containers hosts is isolated and encapsulated virtualization software that is run within its own address space. It packages control group, namespaces, bins/libraries and other dependencies for hosted applications. It effectively virtualizes the host operating system by leveraging the kernel [74][75].

Deploying multiple containers or virtual machines to implement applications or networking node is an essential functional requirement in the development of the prototype. This process can be optimized through orchestration, becoming increasingly valuable the larger the number of virtual environment become. The subsequently mentioned PaaS solutions, include a number of features, including: Instantiating of environment and provisioning of hosts, scaling clusters, re-scheduling environments, automating networking, e.g. creating (SDN) overlay networks and exposing external plug-ins and services. Each of these provide different feature sets in regards to the services that can be supported in relation to virtual machines and containers, but share many of these same high-level features, e.g. all of the presented platforms can employ OpenStack. Each platform has its own respective learning curve.

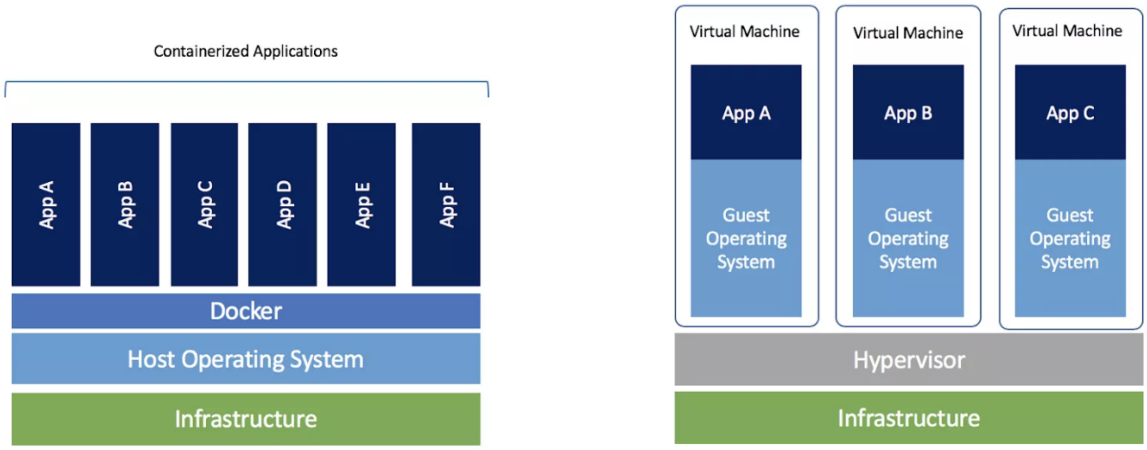

Figure 2.16: Architectural comparison of containers and virtual machines [76].

- **Docker swarm** - A Docker native platform that is tightly integrated with the Docker API, making it well-suited for use with Docker CLI. The same commands that applied to a single host docker cluster may be employed with Swarm [13].
- **Kubernetes** - A portable, extensible platform for managing virtualized workloads, e.g. micro (Kubevirt) VMs and containers. It has a large, rapidly growing ecosystem. Kubernetes APIs are widely available. [11].
- **OpenShift** - A platform that offers a layered architecture that includes and builds upon Docker clusters and Kubernetes. These assist in deployment of lightweight LXC-based containers and images, where Kubernetes assists in the orchestration and management across hosts [12].
- **CloudFoundry** - A Platform where applications are deployed, scaled and maintained by BOSH. It employs Docker images, buildpacks and cloud controllers to quickly deploy versioned software in a virtual environment [77].
- **MESOS** - A platform designed to scale very large clusters. Mesos supports diverse kinds of workloads such as Hadoop, Kafka and Spark tasks. Can be extended through Marathon, Mesosphere DC/OS, Chronos or Apache Aurora [32].

Chapter 3

Analysis

Chapter 2, the State-of-the-Art, presented a series of architectures, frameworks, specifications, protocols and technologies, which are either core theoretical or mechanical concepts within the SDN, NFV and cloud-based networking fields or concurrently being employed by CSPs. These concepts are required to elicit possible ways to construct and interconnect a distributed emulated testbed for SDN controller and OpenFlow implementation and testing. Penultimately, this implementation may be embedded as part of a software solutions that can be readily available for employees, researchers or students working with SDN. Thus indirectly being leveraged by CSPs or other networking enterprise in facilitating the conversion between traditional and software-defined network architectures. Ultimately looking to solve for technological, organization and operational challenges mentioned in chapter 1.

3.1 Service Architecture Development

This section seeks to analyze the candidate technologies, or components therein, among the proposed concepts that may be feasibly employed for an initial proof of concept deployment. It further provides an limited assessment on how these candidates or components may be compounded in a conceptual design for subsequent implementation. The full description of the proof of concept prototype is presented in the conceptual design and implementation chapters - chapters 4 and 5 respectively. These processes are presented in a step-by-step approach, where each subsection connects to the next and summarily provides an overall analytic perspective on how a potential architecture could be structured. The chapter concludes in an associated requirements specification, derived from the analytical process, stipulated scenarios and industry interviews.

3.1.1 Virtual environments

As the prototype deals with computing environments and emulating network service functions, there are certain consideration that needs to be made in regard to the utilization of hardware resources and minimizing virtual overhead, in order to be able to provide an as accurate representation as possible of real network performance scenarios. To that extent, this section presents an analysis of containers and virtual machine workload environments and how these can be used in unison with network emulators presented in the subsequent section.

Between virtual machines and containers, containers have lower resource overhead through not requiring operating system images for operation or hypervisors for instruction translation. Container images are still required, but these are typically lightweight and may be easily shared and layered between container hosts, allowing them to re-scheduled or terminated quickly and efficiently reproduce research or development results [78][79]. The mentioned parameters naturally also affect boot or startup times for the respective environments, which is an elemental concern when considering scaling cloud native applications [80]. Both Virtual machines and containers have a high degree of portability between operating environments, providing both ample opportunity to be moved between development and production, public and private cloud, environments. Although virtual machines may require more setup time and complex orchestration methods in comparison to containers [74][75].

Container technology provide flexibility and efficient when deploying and scheduling lightweight software functions. This have lead to the employment of containers having garnered increased attention and development efforts from both market and research actors [74]. While the mentioned parameters make containers seem like the ideal candidates for the type of prototype being developed, i.e. deploying network emulators and network function virtualization, the workload performance between the two separate approaches still needs to be investigated.

Zhange et. al. presents a comparative experimental study between virtual machines and Docker containers using an Apache Spark environment [81], measuring deployment convenience, boot-up efficiency and application performance. They concluded that setup behind constructing clusters are shorter for container due to image sharing, noting that this is a soft value depending on the tester. More importantly it was concluded that with different workloads containers elicited better scalability in terms of boot and execution time, while with the same workload containers achieves higher CPU, i.e. I/O performance and memory write utilization, but notes that the difference in overhead between virtual machines and containers is negligible. Conclusions that are supported in similar studies carried out by Morabito et al. [82] and Bhimani et. al. [83].

One approach to bridging this gap, could be to employ mini virtual machines, e.g. KubeVirts. However, these do not achieve the same widespread integration as Docker containers has in the solutions mentioned in section 2.8. It is due to this integration into open source solution that Docker containers make a strong candidate. These were chosen

as the go to multi-host networking option for the proof of concept prototype, partly due to being able to scale out CPU and memory resource availability over a IaaS infrastructure and partly due to the support for interconnection options presented by Docker.

A decision that is supported by the fact that many of the network emulators presented in section 2.7 make use of LXC and Virtual Ethernet (Veth) pairs for connecting emulated nodes in different Linux networking namespaces. The Linux namespaces provide isolation for running processes, e.g. switches, controllers and hosts. This should allow for Docker hosts to be employed for any given virtual network function process [84]. A similar approach is proposed by Xu et. al. [85].

This decision is further supported by Docker containers being able to support Virtual Extensible Local Area Network LAN (VXLAN), similarly to OpenStack [14], for creating layer 2 overlay networks over layer 3 infrastructure [13]. These options should allow for the connection of distributed networking root namespaces (switches and controllers) and namespaces (hosts) either through Veth and or the use of V(X)LAN functionality for the creation of SDN overlay networks and the establishment of links through Point-to-Point Protocols (PPP), e.g. GRE, PPTP, LT2P - features that are required for setting up and connecting a distributed containerized emulation environment.

Using Docker containers, it further could be argued that to employ Docker Swarm as the primary orchestration solution for the project would be an ideal approach within the available time-frame, as the commands and APIs are similar, while also being integrated with Docker Engine and Compose. Furthermore, the swarm manager could be employed to automatically assigns addresses to deployed containers on a created overlay network when it initializes or updates the containers, thus saving time on setup and boot time. Lastly, DockerHub allows for easy distribution of images between these deployed containers, allowing for images to be reused to emulate network functions.

3.1.2 Network emulation

As mentioned in the chapter introductory paragraph, the core tenants of the elected approach seeks to minimize virtualization overhead and learn-ability and as such this weighs in when selecting a open source SDN-capable network emulator. While remote procedure calls and multi-instancing are part of the consideration for all of the listed network emulators, there is a possibility for implementing these features, if they are not part of the native suite of features, either through third party solutions. It would however increase implementation time. As such these features weighs in lessened degree. As summarized on figure 2.15, each network emulator have different approaches to supporting and managing communication overhead, node virtualization, link emulation, remote control capabilities and availability and openness of API libraries. Furthermore, the decision to employ Docker containers also need to be taken into consideration. All of the presented emulators presents some degree Graphical User Interfaces (GUI), which can simplify topology presentation and modification and lower barriers to employment of UNIX-based systems.

In a emulated virtual testbed it may be required to perform duplicating operations on

packet flows between virtual nodes or kernel and userspace, in order to simulate switching behaviour. This methodology can be found in the CORE (implementation-specific - may be omitted) and NetKit emulators. Logically, this detail incurs a computational penalty in terms of overall CPU and memory performance. Furthermore, the representation of different isolated and encapsulated network nodes may be achieved by different virtualization techniques. NetKit employs virtual machines, e.g. full virtualization to generate separated networking nodes, this incurs a penalty similar to that described in the paragraphs concerning virtual machines, i.e. higher CPU and memory usage, thus limiting such approaches to be employed for testing smaller network topologies. IMUNES, CORE and Mininet leverage compartmentalization techniques on operating system kernels and network stack virtualization to constitute separate emulated networking nodes within a emulated testbed. These approaches have less overhead and as such have a lessened impact on CPU and memory utilization [86]. On this basis NetKit is discounted.

IMUNES presents a monolithic virtualization library with limited options for changing classes and functions and initial probing showed it to be too comprehensive to modify to the extend needed to fit the requirements of the proposed prototype, within the allowed time-frame of the project. The CORE emulator is a fork of the IMUNES emulator and can employ FreeBSD containers, as IMUNES, or Linux Containers (LXC) network stack (namespace) functionality for virtualization of networking nodes. The apparent issue with employing the latter option together with Docker containers is that this implementation has duplicate communication overhead. which would as previously stated be an impairment to the overall emulated performance. Mininet was ultimately chosen as the employed network emulator candidate as it is directly developed towards deploying and testing SDN networks and because it includes standard UNIX/Linux network applications, i.e. Linux kernel and network stack. As such the network namespaces that are similar to how Docker manages its, allowing these to be used as isolated and encapsulated emulated hosts. The Docker Swarm COE allows for rapid deployment of multi-host networking as a result. Ultimately, this should allow for custom topology scripts and configurations to transferred to production environments [46].

Further research also suggests that Mininet is able to perform accurate performance modelling that is considered to be adequately suited for performance experiments in SDN and OpenFlow architectures, as it has been used to reproduce a larger series of research results within the field of SDN [87]. This has subsequently lead to significant support within the open source and research community (ONF) [88], especially in regards to the development and verification of the performance metrics of SDN controllers and architectures. This is in large part due to the open extensible CLI and Python API library and modular nature of the Mininet classes, methods, functions and variables. Together with Mininet's employment of Linux networking namespaces, this should prove highly beneficial for patching Mininet with Docker. This can further provide exclusive details on links, e.g. addresses, interfaces and ports, and routing tables of each container host.

Furthermore, this should enable the creation of python automation scripts that can is-

sue commands using the Docker and Mininet CLI when a specific class method is invoked. The Mininet API further enables the implementation of custom provisioning scripts [89]. These can be configured via a text editor, potentially either via a web server or manually. Regardless of implementation, these can be free distributed and booted in separate environments.

Additionally, related work indicate that initial steps has been taken to support for distributed network environments and clusterization across several physical hardware machines, as suggested by the concurrent state of Mininet Cluster Edition [90] and Maxinet [91]. As mentioned, while these focus on deploying Mininet across multiple physical machines, they present a logical foundation on which to improve and be inspired, as part of the process of building a distributed emulated testbed for deploying and testing SDN controller and Openflow setups, performance and behaviour. It would be possible to use separate process spaces with Linux containers (Docker), but currently Mininet does not do that [46].

3.1.3 Network Function Virtualization

As previously mentioned, the Docker Engine employs container-based virtualization to create multiple isolated container that can run natively in most environments. Each of these containers can be employed as hosts to function as encapsulation environments for emulated processes. These may then be connected by Virtual Ethernet (Veth) pairs separated by hierarchical levels (0-2) of namespaces. This is achievable by extending the Mininet API, creating the option to boot up Docker containers as Mininet hosts. Because the data layer is encapsulated in containers, it should be feasible to employ IaaS solutions to automatically scale the available resources needed to dynamically create variable-size topologies. DockerHub allows for easy distribution of images between these, enabling images to be reused across containers to create emulate environments and reduce overhead, as per section 3.1.1.

The Docker and Mininet API libraries can be employed to respectively emulate and configure a series of switches that can be used to constitute a switching fabric, e.g. tree and star, as well as simulate traffic-generating hosts and parametric links (performance modelling), for a given compatible SDN controller - discussed in chapter 5. It can be argued that these will function as a micro-service architecture, where the distributed emulated processes can be deployed inside isolated and encapsulated containers that can be on a IaaS solution, as previously mentioned.

While it is not employing the NFV architecture, it builds on many of the same principles, by deploying emulated network functions in Docker containers over a general infrastructure - It employs containers instead of virtual machines to host emulated network functions. In prospect, these can potentially be employed by SDN platforms to build Network Service Chaining and Service Chain Provisioning setups or develop and test SDN applications and controllers over an emulated data layer. This can eventually be used if or when the functionality of the proposed proof of concept prototype can be extended to

include additional network functions such as routers, firewalls and load balancers - the Mininet API would have to be extended substantially to support this.

Similar approaches for employing containers to enable improvements in provisioning and performance of VNFSs have been proposed in research literature. For example, Cziva et al. presents a series of studies on how container-based NFV for SDN can be employed for switching networks [92][93][94]. While these approaches are primarily aimed towards the creation of virtual middleboxes for SDN networks, a similar perspective can be adopted for constructing and provisioning network functions over a distributed container architecture within an emulated testbed.

It is plausible to expose Mininet to the internet by adding a new interface to a VirtualBox VM as a bridge adapter, allowing it to communicate with a DHCP server on a LAN or by making a bare-metal installation on a local host or IaaS. This should in theory enable any Mininet host on a Docker container to communicate with external service, although enabling Docker networking for Mininet hosts breaks emulated link functionality. A plausible option around this problem would be to define Python modules that could translate API end-point requests and responses to provide PaaS and IaaS solutions the capability to interact and manage with Docker containers inside the emulated testbed, enabling the option to integrate NFV solutions such as OpenStack.

Such PaaS implementations, as mentioned in section 2.8, and solutions such as OPNFV and OSM, can be employed to set up VNFs, based on either OpenStack Quantum or Neutron API configurations together with services such as OpenStack Nova, Heat, Keystone or Glance to establish Points of Presence (PoPs), as it allows for virtual machines to function as hosts, while also establishing either VXLAN or GRE connections between these [14]. In regards to the proposed proof of concept prototype, inspiration has been taken from how these can be employed as orchestration platforms to provision and manage PoPs and support SDN controller platform capabilities.

3.1.4 Switch Specifications

Both SDN data plane switches presented in section 2.2 are featured with programmable OpenFlow interfaces that can be exposed to the SDN controller platforms presented in 2.3. With respect to the previous section, Mininet typically employs a default Linux bridge or Open vSwitch (OVS) [43] running in either kernel or user space mode, depending on the use case, for network packet switching. There is still another open source switch that requires mention, i.e. the Basic OpenFlow Userspace Software Switch (BOFUSS) [95]. Both OVS and BOFUSS have unique properties and features that make them suitable for different types of use cases in terms of constituted production and research objectives. On one side, the OVS is the most popular industrial grade virtual and physical software-defined switch, which is concurrently mostly seen being employed in SDN-capabable datacenters and to a certain extend carrier-grade networks. On the other side of the spectrum, BOFUSS is popularly employed as a software switch in regards to fulfilling open source research objectives, controller evaluation, and proof of concept implementations within the field of

SDN, as it, i.e. the datapath, has been specifically designed towards leveraging OpenFlow principles and features simpler code [47]. BOFUSS has been developed iteratively, forked and named different things throughout various research publications, i.e. CPqD switch, ofsoftswitch13 (GIT) [95], OF13SS and OpenFlow version 1.3 software switch, Ericsson TrafficLab 1.1 softswitch and Stanford OpenFlow 1.0 Userspace switch [47].

As briefly touched upon above, there is a general consensus that OVS elicits better performance metrics at the inherent cost of being more constrained in terms of modification. Where BOFUSS provides a simpler code, but with more options for extensiblility and cutomizability through the separation of available API libraries needed for more diverse SDN controller and OpenFlow implementations - an example implementation is provided in figure 3.1. As such, the basic BOFUSS elicit reduced performance metrics in comparison. These are typically the differences being highlighted. Evaluations that are often done in relation to the base versions of BOFUSS and OVS and here it should be noted the BOFUSS features a wide range of extensions to both the switch itself and the OpenFlow protocol. This customization aspect is further one of the factors weighing in on the overall consideration made on the subject of software switch capabilities.

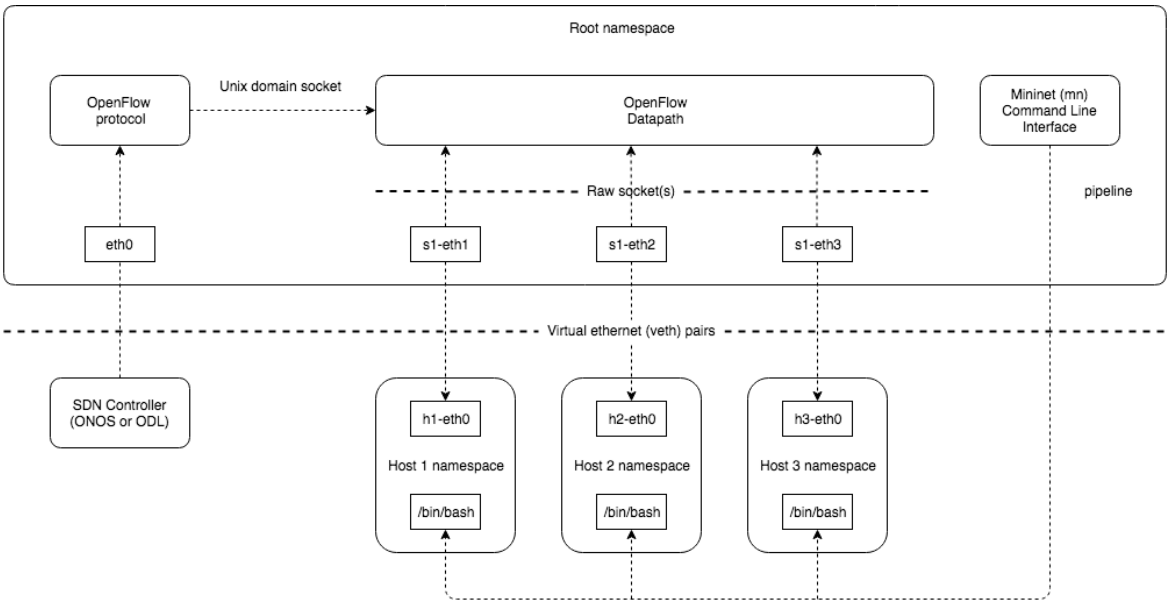

Figure 3.1: Mininet BOFUSS namespace implementation - inspired by [96]

Fernandes et. al. [47] presents a comparative evaluation between OVS and BOFUSS, which further take one of the more developed extensions into consideration as well, i.e. BEhavioural BAsed (BEBA) forwarding [97], which is a prototype that is built on top of BOFUSS. This extension provides stateful packet forwarding and packet generation that are based on the OpenState project [98]. In short, this effectively means that stateful SDN

application components may be embedded in the data plane networking nodes, resulting in BOFUSS being able to dynamically adapt the forwarding policies and its actions based on ingress instruction field events, thus changing the overall flow state by matching it the respective flow tables [98]. This does however require initial setup through the associated SDN controller and its respective interfaces. This feature builds on the functionality illustrated in the flow diagram presented in section 2.1.1. Additionally BEBA has brought some performance improvements to the BOFUSS data plane that nominally works in the userspace [47]. This BEBA example provides excellent insight into the customizable nature of BOFUSS and how its pipeline processing can manipulated to leverage different features or performance enhancements. As such, BOFUSS can be a good candidate for specific scenarios production scenarios as well, where OVS may prove too complex to operate [99].

The comparative evaluation between these OpenFlow logical software switches is based on two tests; Individual bench-marking of OVS, BOFUSS and BEBA using iPerf [100] and performance metric evaluation of a data center scenario with constituting traffic, where each network is comprised of the respective switches.

The first benchmark test of the three individual software switches showed that OVS can sustain a throughput that is significantly higher than both BOFUSS and BEBA. OVS can support a throughput of 51,413 Gbps, where BOFUSS can support 0,186 Gbps and the enhanced BEBA can support 1,184 Gbps. As the results indicate, this supports the common notion mentioned in the above paragraphs. Fernandes et. al. states that it is important to notice how the BEBA switch surpasses 1 Gbps - a result that is considered a reasonable throughput for a larger group of networking scenarios [47]. The second SDM data center deployment scenario test illustrates a possible switching fabric within a given presented network infrastructure - a spine-leaf topology. Where a traffic generator, with two distinct flow distributions, i.e. data mining and web search patterns, that has been derived from data center network traces [101], was employed to test each of the switching capabilities. It presents a random distribution function of the cumulative distributions and associated payload size classifications (small to large), and illustrates the difference in measured throughput and flow completion time at increasing network loads. The results indicate that OVS and BEBA perform and provide similar results for medium and large payloads, where OVS outperforms BEBA at smaller payloads. In all cases, the BOFUSS is outperformed by OVS and BEBA [47]. Conclusions supported by Tantayakul et. al. [102].

This comparative evaluation proves that BOFUSS extensions can attain improved performance and as such can be implemented in proof of concept prototypes without being hindered by performance of the basic BOFUSS. For network-limited experiments carried out on a single computer, BOFUSS may also be utilized without extensions, as a collection of software switches share CPU and memory resources. It is suggested in such cases to limit link capacity to 10-100Mpbs - well within the limit of BOFUSS.

As mentioned in section 1.7, this is thesis takes an exploratory approach to understand and learn how to employ SDN application and OpenFlow mechanics within a SDN architecture. As such there is merit in employing the BOFUSS for learning to utilize OpenFlow

and SDN controllers, as these are ingrained in its inherent software architecture, and test the feasibility of the chosen implementation strategy, before delving into learning and integrating the more complex OVS. Such an approach should also prove that both software switches can be employed in the prototype.

The Mininet emulator does provide support for the older OpenFlow 1.0 userspace switch from Stanford. Since BOFUSS is an upgraded version of said switch it is possible to install and run it within the emulator with minimum workaround [103]. To avoid confusion, the ofsoftswitch13 github code repertoire is intended for initial reference implementation for the proposed proof of concept prototype to test whether the containerized distribution approach for Mininet images could be employed - This code base is the same as the base version of the BOFUSS.

3.1.5 OpenFlow and SDN Controllers

As with OVS and BOFUSS, each SDN controller presented in section 2.4 has varying levels of support for the different versions of the OpenFlow protocol - ranging from 1.0 to 1.4, where 1.5 is largely unsupported or in the experimental stages of development. Only the latest version of OVS (2.8) has default support for 1.4, where 1.5 has to be user-enabled and has multiple missing features [43]. Mininet concurrently only support 1.3. Kspviswa presents an illustrated overview of the suite of features implemented across the different version of OpenFlow [104]. The baseline version that is safe to employ across both OVS, BOFUSS and the majority of the presented SDN controllers remain 1.3. As a result that poses a minimum functional requirement for support within the proposed prototype.

As illustrated in figure 2.9 and discussed in section 2.1 and 2.4, because the Northbound APIs are tied to the application layer, they must provide support for a wide array of application and their associated network requirements. Therefore, each of the Representative SDN controller platforms have Northbound REpresentational State Transfer (REST) APIs. This is a logical approach as these presents the abstraction of network services and functions and interface exposed to SDN application seeking to communicate their network requirements to the SDN controller platform(s). While REST does impose architectural standards in terms of the data formats and operations that can be employed by said platforms in the facilitation of those requirements and services, there are still different functional groupings of SDN controllers, i.e. variety of vertical functionality provided by a SDN controller, from different organizations and vendors, some of them proprietary, and instances where the API is required at different hierarchical level. This is perhaps best illustrated by figure 1 in the ONF Nortbound Interface charter for SDN Controller platforms [105].

As illustrated in figure 2.9, the Southbound API in SDN controller platforms are significantly more varied in terms of supported OpenFlow protocol versions, the concurrent network monitoring and configuration protocols presented in 2.7.1 and 2.7.2 respectively, and more. Summarily this at odds with SDN and NFV principles until API consolidation occurs, as it is difficult to code against. Furthermore, it poses significant challenges in

regards to the envisioned expected outcome.

Each of the different SDN controller platforms described in section 2.4 present different advantages and disadvantages, as they are built towards fulfilling requirements within different scenarios and use cases, i.e. datacenter, WAN, carrier, research or campus. Entailing that some are better suited for production environment, while others are better for research and development. There is not one SDN controller platform that is best suited for all use cases, an argument based on the various degrees of network functions and service abstraction layer functionality provided, e.g. plug-in management, capability abstractions, flow programming and mapping, etc in each of the SDN controller platforms.

As the inherent goal is to presents a proof of concept prototype and the thesis focuses on developing a modular approach to implementing open source SDN controller platforms over an emulated switching topology, any of the described SDN controllers may be employed in a test scenario. However, for initial implementation, there are a few SDN controllers that stand out. These are the Pox and ONOS controller. The decision for these controllers to be part of the initial development of the prototype is substantiated by the degree of open source community support and concurrently available documentation. Also, POX is one of the SDN controllers that are natively supported in Mininet, which allows for easy testing during the prototyping phase and that ONOS and Mininet are both supported by ONF and may therefore be easily connected using the ONOS.py library [106]. Further, POX is designed for centralized SDN architectures, where ONOS is designed for distributed, providing an interesting perspective in terms of implementation. Both can be in accordance with a provisioned Mininet network topology.

The limit of two SDN controllers, for initial implementation, is based on the notion that it is a proof of concept prototype, where two was deemed a substantial number to prove the functionality of said prototype and save development and implementation time. An approach for providing insight into the workings of SDN controller platforms is to look at the behaviour of different SDN controllers and how these operate within specific classes of topologies and workloads. How the different operating characteristics and statistics of the respective SDN controller within different scenarios can be extracted is discussed in section 3.2.

As described in section 1.6 and above, the expected outcome was to be able to support different SDN controller platform templates that a given user could deploy easily within a custom or standard topology. While Mininet does support a wide selection of these controllers, these do however need to be configured individually to interconnect properly with the emulated container hosts, i.e. where the SDN controller can support dynamic network programmability by providing templates that enable the creation of scriptable CLIs, these are still required to be setup accordingly. There are two possible approaches to solving this challenge and the challenges associated with Northbound and Southbound APIs described above; Change Mininet custom topology scripts or as previously mentioned, individually configure the SDN controllers in the emulated development testbed once this has been booted. An approach to solving these challenges could be to provide

standard topology templates, by booting previously configuring custom topology scripts from a database. These would however need to be re-configured if a change would be to be made to the switching fabric or any of the associated protocol parameters. This same feature should allow developers to transfer configurations between different computers or cloud environments.

3.2 Network Monitoring and Presentation

As observations on network parameters from different data sources is the basis for analysis and subsequent decision making processes. This section seeks to analyze how network monitoring may be performed and how features for SDN controller platforms and OpenFlow interaction may be presented to a user in a detailed overview. This is carried out in order to display the behaviour of a given SDN-enabled network over an interval period of time and to provide insight into the performance and interaction, while taking the selected software components into account.

In general, there are a few different approaches that can be taken in eliciting observations, i.e. parameters and features from the different data sources available within an emulated SDN network needed represent network statistics to a user. Observations represent raw data that is retrieved from from SDN controller platform northbound and southbound APIs, extensible network monitoring protocols and packet capture and data analysis tools. An example of such observations could be flow table entries at a specific node within the provisioned network or the current transmission count of packets at a given switch port.

There a many different data sources that can be interacted with, but as the proposed proof of concept prototype is centered around dynamically provisioning and implementing SDN functionality within an emulated environment, the Northbound and Southbound APIs of the selected candidate SDN controller platforms and network element agents will logically be considered the primary sources for deriving network observations.

Any SDN application can employed the Northbound REST API of the constituted SDN controllers. However, to further support the development of SDN applications using the emulated testbed, a monitoring application may be developed that would continuously execute queries to agents in networking nodes, in order to elicit concurrently available measurements on the monitored network. These observations would need to be made available to the SDN applications in question. In regards to vendor-neutral network monitoring protocols and technologies mentioned in section 2.6.1, there are a few considerations to be made

Based on available support, its capabilities to employed both as a network monitoring and configuration protocol and its support in switches, SNMP looks to be an ideal candidate. It is however currently facing incompatibility issues with OVS kernel switches in Mininet. Where the functionality provided by port mirroring can largely be carried out by the respective integrated SDN controllers and Mininet functionality.

In regards to the sFlow protocol, this may be integrated with sFlow-RT [107]. An an-

alytics engine that can aggregate and expose network metrics through REST APIs. This could potentially supplement the SDN controller REST APIs for the development of SDN applications. While IPFIX provides similar methods of exporting flow records to monitoring solutions, logging flow records may be considered limited in relation to the functionality provided by sFlow and sFlow-RT.

As the Northbound and Southbound APIs of the different SDN controller platforms varies in their implementation, i.e. various interfaces exist in different places throughout the SDN control stack, it is possible that distinct functions may be needed for some. However, as the Northbound APIs are all defined as REST interfaces, it should be feasible to construct a common implementation that can measure a core set of performance measurements and management information across controllers. Here, it should be noted that many of the SDN controller presented in section 2.4 already have GUIs in place that represents these measurements, i.e. flow, port, group, meter and queue statistic services.

Similarly for the Southbound API, it can be noted that all of the chosen SDN controller platform and Mininet support Openflow 1.3, which should allow for common implementation to be defined across these. Working with one protocol version makes it inherently easier to work with the Southbound API, and together with the Northbound API, also to query for aggregate network node, port, link, flow and packet statistics. Where the link is the connection between two ports where a flow may be active.

A further way to get insight into the behaviour of the network would be to employ packet capture and data analyzer tools, i.e. Wireshark as it has an OpenFlow dissector extension. This can possibly be run on the side, but it is likely that a wrapper would be required to implementing this functionality, unless a Secure Shell (SSH) can be established inside the Mininet environment. Additionally, such tools require an initial selection of network parameters before they can be initialized properly and subsequently produce large amount and scalability issues can be stipulated [108]. These two approach are also not directly integrated in SDN architectures, meaning that they can not exploit the benefits of a centralized logical controller, but they may provide network-wide observations.

These observations may be presented in the same form as they are collected, but commonly, granted the volume of the representative data, it would be necessary for this associated data to be be pre-processed, parsed and categorized in order for it to make coherent sense, i.e. mapping multiple measures or sampling time stamps into an aggregated format, such as per packet or per flow. In-line with web-based SDN controller GUIs and available REST APIs, this may be achieved through a separate web application centered on JavaScript (Node.js), HTML and CSS with jQuery and Bootstrap shortening deployment time. Its immediate purpose would be to call a pre-defined set of requests and display or store the results in a data structure format that can be easily presented or manipulated further. Lastly, as this data should be presented in an easy to understand manner, there will be a need to present the data in different report formats, e.g. per flow or intervals, to facilitate visualization of the data. A variety of list views and graph tools can be employed to display changes over time. There are different open source libraries that may assist in

the development of the above-mentioned.

To provide a more dynamic view of the network components, it may be feasible too provide an individual view of the representative nodes within the network, by letting a user choose one of the nodes in the topology to expand and show its configurations, associated statistics and how the node-specific performance is in this given case. If the time permits, implementing a comparative feature for the SDN controllers, would provide valuable insight.

3.3 Industry interview

In an interview with Motorola, it was described that both traditional network hardware and virtual network functions are employed. In the case of the former, a network management system is employed to perform network configurations of external ports and addresses. In the case of the latter, VMware vSphere [109] has been employed to perform server virtualization, where virtual server has an isolated logical switching fabric for connecting provisioned virtual machines.

Motorola has brought a subset of services into the cloud, where Kubernetes together with Cilium [22] are employed as the SDN solution. The SDN controller is implicit in this architecture, as in controllers certified by Kubernetes may be chosen. The functions in these are defined via APIs and policies. Considerations regarding the integration of Juniper contrail is currently underway. In most cases the cloud is employed for testing, with less emphasis on local testing. For local testing, MiniKube [110] and MiniShift [111] is utilized to run Kubernetes and OKD clusters locally. Similarly, the SDN controller is a constituent part of these solutions.

In summary, focus is not on the SDN controller performance modelling, but on the provided functionality, i.e. suite of features that can be supported for functional modelling of services. In Motorola, SDN is employed to facilitate and orchestrate connectivity between application components in a micro-service architecture, as the number of interfaces and links increase drastically. The migration from monolithic applications to micro-service applications is noted as being a tremendous task.

An identified issue is that debugging and logging output is typically forwarded to the employed provider, e.g. Microsoft Azure and insight into the performance and interaction of SDN controllers may be limited as a result thereof.

3.4 Inspiration

As touched upon and as presented in a review of integrated SDN and NFV architecture by Bonfim et. al. [39], there are wide differences in implementation approaches among the design architectures concurrently being put forth. Research into testbeds for SDN architectures revealed that solutions such as OpenSDNcore [56], SoftFire [112] and AARNet [113] expose very little that can be employed. SoftFIRE, does however provide limited

middleware that enables Docker container provisioning using Docker Compose [114][13] and OpenStack[14].

As these testbed solutions are closed sourced or black box systems, it makes it inherently difficult to take these into consideration. The only open source git repositories available based on secondary research, were that of the network emulators discussed in section 2.4 and 3.1.2. As such these where looked to as the foundation for constructing an emulated testbed. Research studies put forth by Cziva et. al. [92][93][94], Xu et. al. [85] and blogs [115][116], served as inspiration for extending the Mininet API through subclassing, altering class variables and parameters and Python extensible CLI commands to interact with Docker.

The differences presented by Bonfim et. al. [39] range from framework design, employed virtualization tools and computing environments, SDN Northbound and Southbound APIs, placement of SDN elements with NFV frameworks and use of SDN controllers. All of these architectures and the orchestration platforms presented in section 2.8, have different general objectives and all have associated advantages and disadvantages as a result [39].

Presenting a prototype that can dynamically encompass all of these approaches is unfortunately beyond the scope of the thesis. Therefore limitations are imposed in section 1.5. As presented in section 1.6, the focus is on implementing an emulated testbed that can dynamically provision data layer network elements and to test and visualize SDN controller and OpenFlow interactions and implementation. Programming the emulated testbed as python class extensions to the Mininet API allows for further implementation or tweaking of integrated functionality. This could be carried out in order to meet requirements of more complex scenarios in a full scale deployment.

3.5 Requirements Specification

The requirements specification section seek to define and prioritize a series of functional and non-functional requirements in order to determine the scope of the proposed system design and architecture. As such, defining system or component functionality in terms of input, processing and output behaviour, as well as imposing system constraints on the conceptual design and implementation. The prioritization is conducted employing the MoSCoW analysis framework. Each of the presented requirements will be numbered in accordance with their requirement category to be used as points of reference in chapter 4 and 5 respectively. Based on section 1.6, 3.1 and 3.2, a number of use cases has been defined to encompass the envisioned functionality of the proposed proof of concept prototype.

3.5.1 Use Case diagram

This section introduces the use cases that has been derived from the aforementioned sections. These seek to outline the lists of actions and events that define the interactions

between the user and the system. These will later be described in terms of functional and non-functional requirements. The flow between these actions and events are depicted in an activity diagram - figure 3.2. The presented activity diagram is further illustrated through a series of more detailed sequence diagrams described in chapter 4.

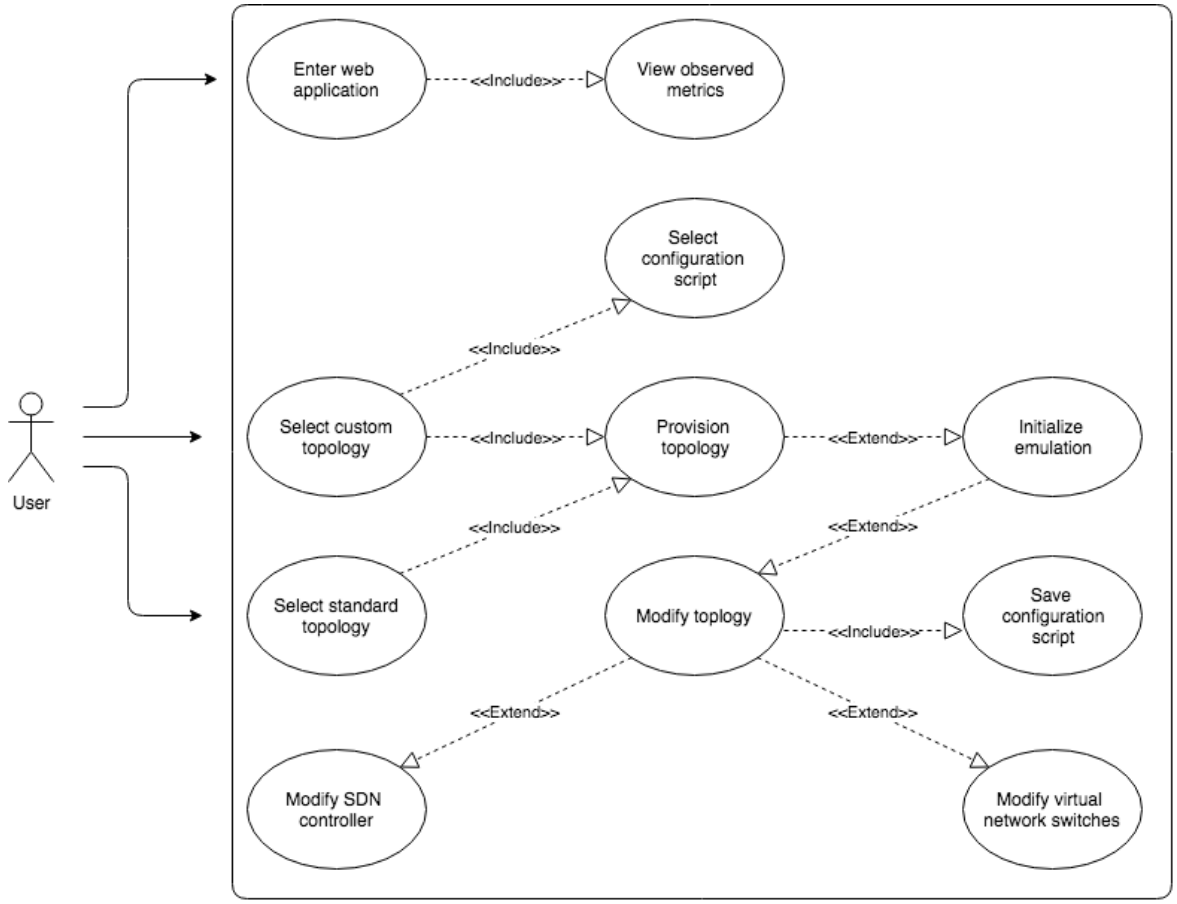

Figure 3.2: System Use Case Diagram

Activity and Flow Diagram

This activity diagram shows the visualized general flow of the proposed proof of concept prototype. The diagram presents two main points of interaction that can initialize the implemented functionality, these are presented as the Dashboard and Topology view menus. These menus point to different groups of actions and associated events. In short terms, the dashboard relates to the import and provisioning of specified topologies, while the topology view relates to the management of the emulated environment.

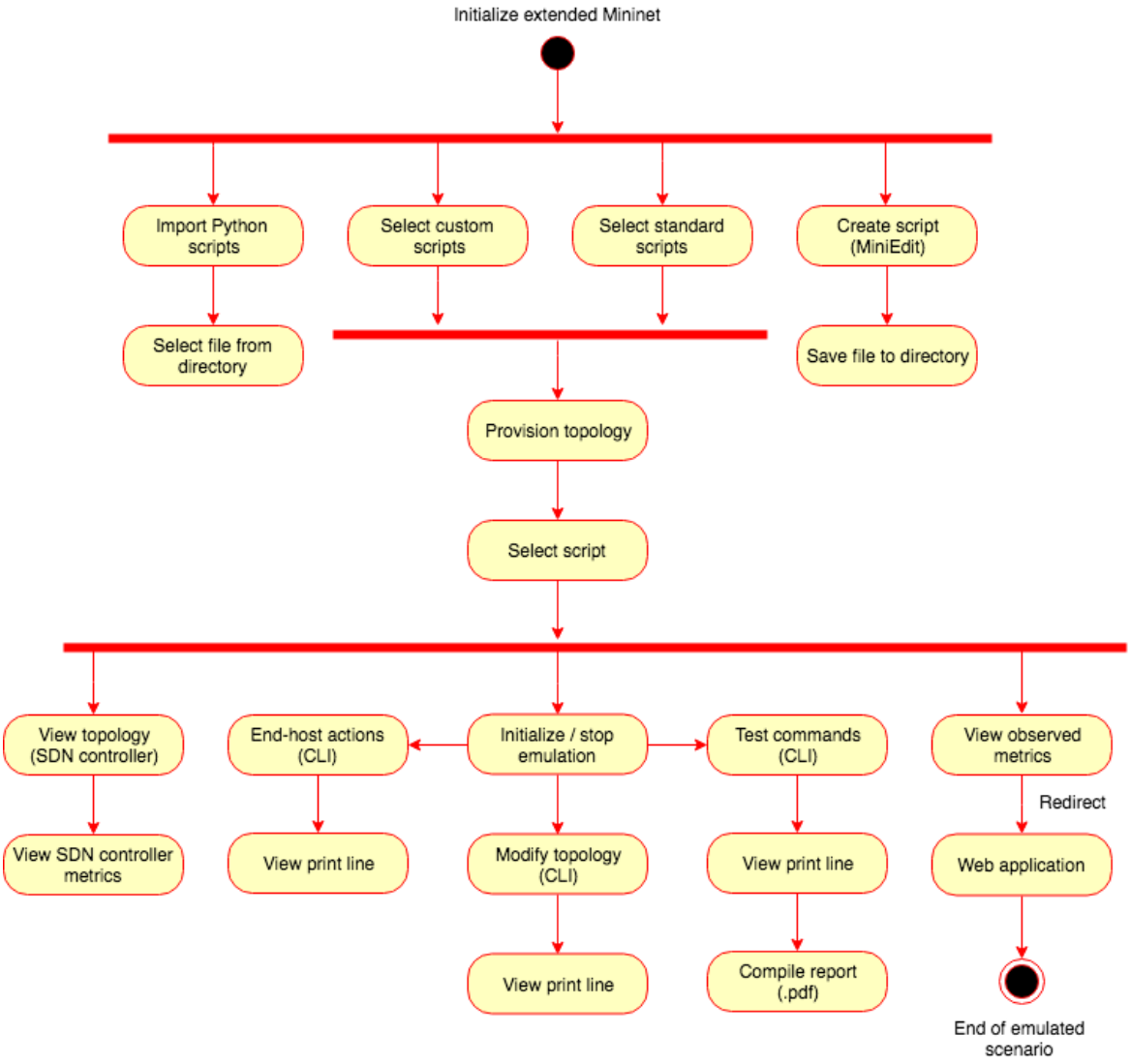

Figure 3.3: Activity diagram detailing the flow of the use case diagram

3.5.2 Requirements prioritization - MoSCoW

MoSCoW is employed due to the timely nature of the project. While the elicited requirements are all important to achieve the intended suite of functionality, they need to be prioritized to ensure delivery of the most immediate requirements and core functions. In accordance with MoSCoW, Must have requirements are the requirements that will be striven to be implemented first, where Should and could are selected in descending order. Should the timescale shift, these will be de-selected in ascending order. Won't have requirement will not be implemented [117]. The described requirements relate to the initial implementation. A full scale implementation would feature additional requirements.

3.5.3 Requirements

This section presents an ordered list of the functional and non-functional requirements that has been classified in accordance with the MoSCoW schema. Each requirement contain a unique name and number, a brief summary and a reference.

Functional Requirements

ID	**FR#1** - The system must be able to create and terminate docker containers.
Description	To provide isolated and encapsulated process environment with low computational overhead, the system must be able to create containers and inherently also terminate unused or run-away containers.
Source	1.6, 3.1.1, 3.4
MoSCoW	Must

ID	**FR#2** - The system must be able to configure Docker containers at run time.
Description	To provide a given user the option to modify a given topology without restarting the emulation scenario, the Docker containers must be able to be configured at run time.
Source	3.1.1, 3.1.2, 3.4
MoSCoW	Must

ID	**FR#3** - The system must be able to employ docker containers as network emulation hosts.
Description	To provide the system the means to emulate cloud infrastructure, it must be able provision new networking nodes at run time.
Source	1.6, 3.1.2, 3.4
MoSCoW	Must

ID	**FR#4a** - The system must be able to connect with networking nodes
Description	To provide network functionality, the system must be able to connect to isolated Linux processes and issue commands
Source	1.6, 2.7, 3.1.2, 3.1.4
MoSCoW	Must

ID	**FR#4b** - The system must be able to establish links between emulated emulated nodes
Description	To inherently provide emulated network functionality, the system must be able to establish links between networking nodes
Source	1.6, 3.1.2, 3.3
MoSCoW	Must

ID	**FR#5** - The system must be able to provision emulated network topologies using template scripts.
Description	To provide the user the options to share and quickly provision network topologies across different machine or instances, the system must be able to import and boot template scripts.
Source	1.6
MoSCoW	Must

ID	**FR#6** - The system must be able to capture and represent network traffic data and control flows.
Description	To provide the user with an understanding of network control and data traffic flows, the system must be able to capture and present these in a GUI.
Source	3.2
MoSCoW	Must

ID	**FR#7** - The system must be able to create and update custom topology template scripts.
Description	To provide the user with an easy way to create and update template scripts, a GUI would provide a better overview of the topology template script.
Source	1.6
MoSCoW	Must

ID	**FR#8** - The system must be able to connect the emulated network with open source SDN controllers
Description	To provide the user with a larger set of options in regards to employed SDN controllers, the system must be able to integrate open source SDN controllers
Source	1.6, 3.3
MoSCoW	Must

ID	**FR#9** - The system must be able to conduct performance modelling over the network
Description	To generate traffic and create specific networking scenarios, the system must be able to conduct performance modelling over the networking nodes and links
Source	1.6, 3.3
MoSCoW	Must

ID	**FR#10** - The system should be able to present a graphical representation of networking nodes.
Description	To provide the user with an overview over the provisioned network topology, the system should be able display the provisioned network topology in a GUI.
Source	1.6, 3.2
MoSCoW	Should

ID	**FR#11** - The system should be able to integrate two or more SDN controller in a single network topology
Description	To provide the user with a options for constructing specific networking scenarios, the system should be able to support multiple SDN controller platforms in a single provisioned network topology.
Source	2.3, 3.1.5
MoSCoW	Should

ID	**FR#12** - The system could be able to be hosted at an IaaS provider
Description	The system requires a UNIX-based OS (Linux) for it to function. Most IaaS providers enables instances to be loaded with most OSs. Therefore, the prototype could be hosted in most.
Source	3.1.4
MoSCoW	Could

ID	**FR#13** - The system could be able to be provide pre-configured SDN controllers.
Description	To provide the user with increased testbed flexibility and modularity, the system could come with pre-packaged and configured controllers.
Source	3.3
MoSCoW	Could

ID	**FR#14** - The system could be able to be integrated with OpenStack
Description	To implement NFV features, the system could be able to integrate with OpenStack and utilize the exposed API endpoints and infrastructure
Source	2.8, 3.3
MoSCoW	Could

ID	**FR#15** - The system could be able to be employed for functional modelling.
Description	The system could be able to support development of SDN applications that may interact with the integrated SDN controllers and provisioned emulated data layer
Source	3.3
MoSCoW	Could

ID	**FR#16** - The system could be able to integrate with the orchestration platforms mentioned in section 2.8
Description	To enable functional modelling, it could be integrated with the orchestration platforms mentioned in 2.8 and their respective feature suites
Source	2.8, 3.3
MoSCoW	Could

Non-Functional Requirements

ID	**NFR#1** - The system components pertaining to Mininet must be written in Python
Description	As Mininet is at the core of the constituted functionality, the system components pertaining to network emulation must be written in a compatible programming language.
Source	3.1.2
MoSCoW	Must

ID	**NFR#2** - The system pertaining to the companion web application must be written in JavaScript
Description	As SDN controllers and monitoring technologies exposes their resources through REST APIs and JavaScript methods, the companion web application must be written in JavaScript
Source	3.2
MoSCoW	Must

ID	**NFR#3** - The system must be able utilize Linux network namespaces and cgroups
Description	To provide the the system with the functionality required to emulate network nodes and integrate Docker and Mininet, the system must be able to support Linux kernel features.
Source	3.1.2
MoSCoW	Must

ID	**NFR#4** - The system must support OpenFlow v.1.3 as a minimum
Description	Based on the OpenFlow versions that are supported by the presented SDN controllers and Mininet support, the system must be able to support v1.3 as a minimum
Source	3.1.5
MoSCoW	Must

ID	**NFR#5** - The system must support Open vSwitch
Description	As the Open vSwitch is considered the de-facto standard within the field of virtual switches, the system must be able to support it
Source	3.1.4
MoSCoW	Must

ID	**NFR#6** - The system must be able to support the Unix shell command-line interpreter or shell
Description	As the system employ the Mininet emulator, it must be able to support the UNIX shell to enable command execution
Source	2.7, 3.1.2.
MoSCoW	Must

ID	**NFR#7** - The system must employ able to use and integrate Mininet and Docker CLIs and APIs
Description	As the Mininet CLI is the at the core of network operations, the Docker CLI should be able to be connected with it.
Source	3.1.1, 3.1.2.
MoSCoW	Should

Chapter 4

Conceptual Design

This chapter seeks to discuss the conceptual design considerations that was made based on the State-of-the-Art and the Analysis chapter. The focus for the conceptual designs is to encompass and fulfill the requirements set forth in section 3.5.3. It lays the groundwork for the subsequent implementation chapter, where each requirement will be highlighted and referred to by their respective identifiers.

4.1 Architecture

Various architectural considerations were made throughout chapter 3. This section seeks to bring these together to constitute a more coherent architecture. The approach to this is to present a series of diagrams that can provide an overview concerning the system terminators, operational flow and structure of the proposed proof of concept prototype.

As discussed, the architectural components of the proposed emulated testbed will consist of variety of virtual and emulated components that has been configured and modified, so that any given user may construct an as-close-to-realistic-as-possible networking topology scenario within a short amount of time. The implementation process is further delineated in chapter 5. In short summary, these components are [89][46]:

- **Docker containers** - Nodes within Mininet API reference architecture that has been tweaked to be represented by Docker containers - Achievable by creating a subclass that can inherit properties and methods from the Mininet object class. In the case of nodes, these are simply hosts in the case of Mininet. As such the Docker containers can be used as Mininet hosts. As Python is an Object Oriented Programming language, objects can be passed around with their respective associated parameters and methods by creating inheritance relations between Node, Host and Docker classes. These node are for a large part OpenFlow-capable switches. This allows for the option initiate a custom Docker class for creating and using Docker containers as Mininet hosts, by calling a series of defined function blocks that are used to set up

the Docker client, pull an image from Docker Hub, create a host configuration, create the containers, start the containers and update the Docker entrypoint and CMD fields [118]. As these have been called, methods in the original mininet.node.NODE superclass can be initiated to create and start shell processes [119]. This enables the core functionality of employing provisioned Docker containers as switches and end-hosts.

- **Isolated hosts** - The aforementioned nodes provide abstractions for interaction with the emulated network elements, that has been distributed across the emulation environment. These distributed networking nodes or hosts are presented in the form of shell processes [119]. The Linux network namespaces and cgroups can be employed for granting these specific network states, i.e. individual or multiple networking nodes may get assigned addresses, interfaces, ports and routing and ARP tables and resource limitations. Interaction can be established and subsequent CLI commands can be sent by using the UNIX shell with Python pipes[120]. A pipe is basically a buffer memory block (filename) in the Linux kernel that may be read from or written to by shell processes [120]. In Mininet, hosts share the root file system, which should make it easier to share configurations [89].

- **Emulated links** - Each networking node can be assigned Virtual ethernet (Veth) interfaces. As with physical patch cables, each Veth pair can connect to virtual interfaces and ports, allowing packets to traverse the hosts. To all configured emulated network elements and SDN controllers, these appear as fully functionally Ethernet ports, allowing for the integration of network monitoring protocols mentioned in section 2.7 and 3.2. It further enables options invoking to measure bandwidth, latency, queues, TCP traffic and CPU usage in Mininet. The parametric values of each of these links, i.e. delay and throughput, is enforced by Linux Traffic Control (TC) function, which may employ a traffic scheduler for performance modelling over the network at configured rates [89].

- **Web application** - The sFlow network monitoring protocol is employed to embed sFlow agents in the emulated network to relay telemetry to the sFlow-RT Analytics Engine. Querying the sFlow-RT REST API, the client-side web application may display network performance metrics in graphs. The web application may further read .csv files exported from Wireshark and present these in HTML tables using client-side AJAX jQuery methods.

- **Emulated SDN controllers** - If installed and configured properly, both the ONOS and OpenDaylight SDN controller platforms can be integrated with Mininet. There are multiple options in regards to how the connection between the SDN controller and the Mininet topology may be connected. The approach taken within the thesis is to employ the Mininet extensible CLI together with the GUI presented by ONOS and OpenDaylight.

- **Emulated switches** - As mentioned in chapter 3, Mininet supports different virtual switches and bridges. The ones chosen are Open vSwitch and the userspace switch, or BOFUSS. These may either be run in kernel or user-space mode. Typically the latter is chosen, as it is easier to modify these, but this does comes at a net performance loss. These may be booted in default configured states and would require configuration to change settings. An overview of the BOFUSS implementation with emulated links, pipes and Linux network namespaces is presented in figure 3.1.

4.1.1 System Context Diagram

Based on the above components, a systems context diagram can be constructed. The purpose of this diagram is to illustrate a high level overview of the boundaries of the system, showcasing the different entities, APIs, applications, solutions and so forth that interact with the system or the system may perform actions with or upon. These boundaries are presented in the form of terminators.

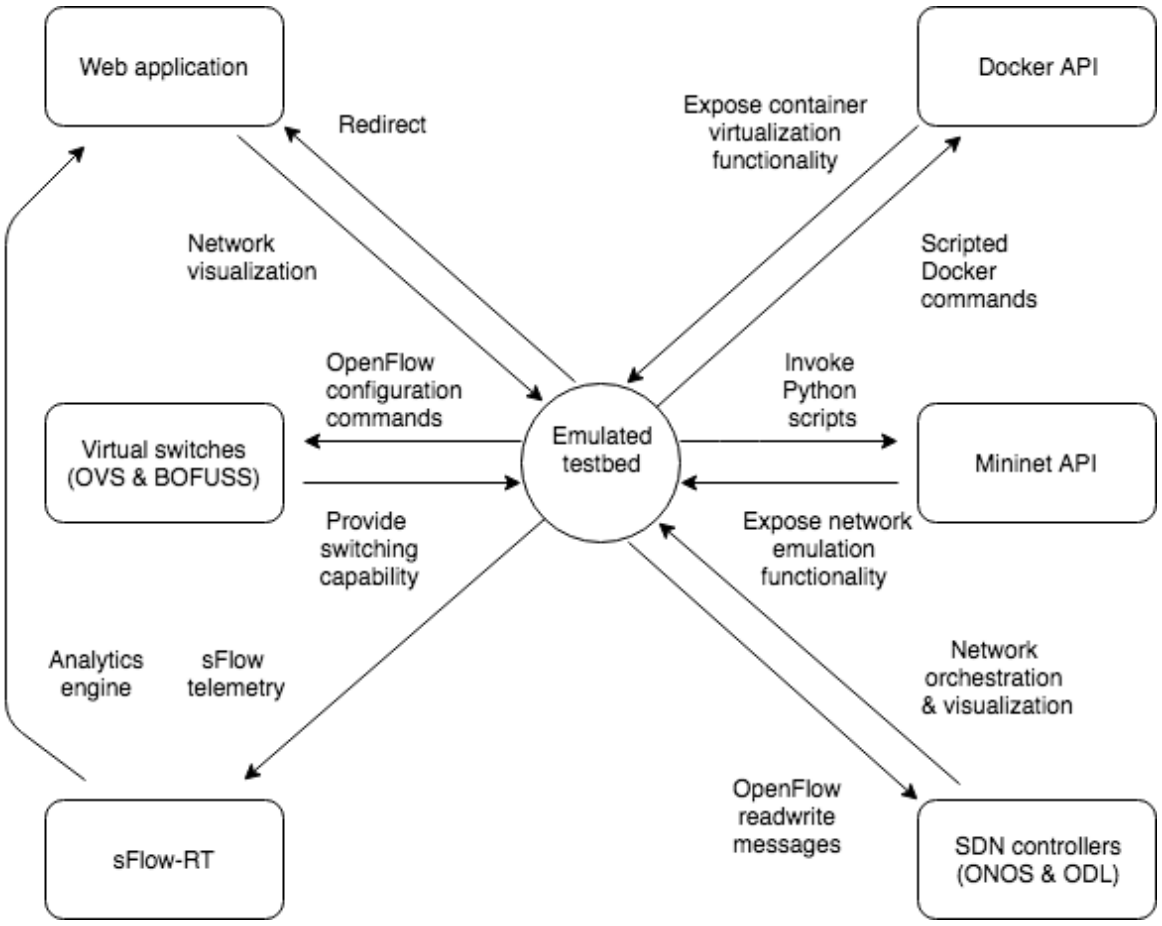

Figure 4.1: Context diagram with system terminators.

Terminators	Description
T1 - Docker API	The Docker Engine API is required for the emulated testbed application to access the Docker Command Line to perform actions on Docker Compose, Dockerfile and Docker Swarm to create and manage Docker containers. The Docker.py [118] Python library for the Docker Engine API allows the emulated testbed to execute Docker commands within the Python application.
T2 - Mininet API	The Mininet emulator API is required for the emulated testbed application to elicit functionality from the Linux kernel. This includes features such as lightweight para-virtualization of networking nodes in kernel and user spaces, access to the Linux Network stack, network namespaces and control groups (cgroups). The Mininet API is supported by a range of Python libraries.
T3 - SDN controllers	There are different options when integration SDN controllers with the Mininet emulation environment. Concurrently it depends on which SDN controllers that are going to be integrated into the Mininet topology. Mininet natively supports the NOX, POX, OVScontroller and RYU controllers which can be structured as custom controller classes, where as other controllers, such as Beacon, ONOS, OpenDaylight and Floodlight, can be configured on a server or local machine and be integrated using the Mininet CLI.
T4 - switches	There are different options when running OVS and BOFUSS switches and bridges. These can be run in either kernel or user space mode. These, as well as end-hosts as deployed as processes run on a Linux kernel. These implement switching functionality through datapaths, virtual Ethernet ports and emulated links, flow tables and so forth.
T5 - sFlow-RT	The sFlow-RT analytics engine receives a continuous telemetry stream from sFlow Agents embedded in network nodes and converts them into actionable metrics, accessible through a RESTful API.
T6 - Web application	The companion web application makes a series of requests to the sFlow-RT API to get access to the Analytics Engine. Based on the telemetry that can be derived from the sFlow agents in the emulated networking nodes, it may display different formats. The web application may further read Wireshark exported .csv files pertaining to OpenFlow control messages.

4.1.2 Sequence Diagrams

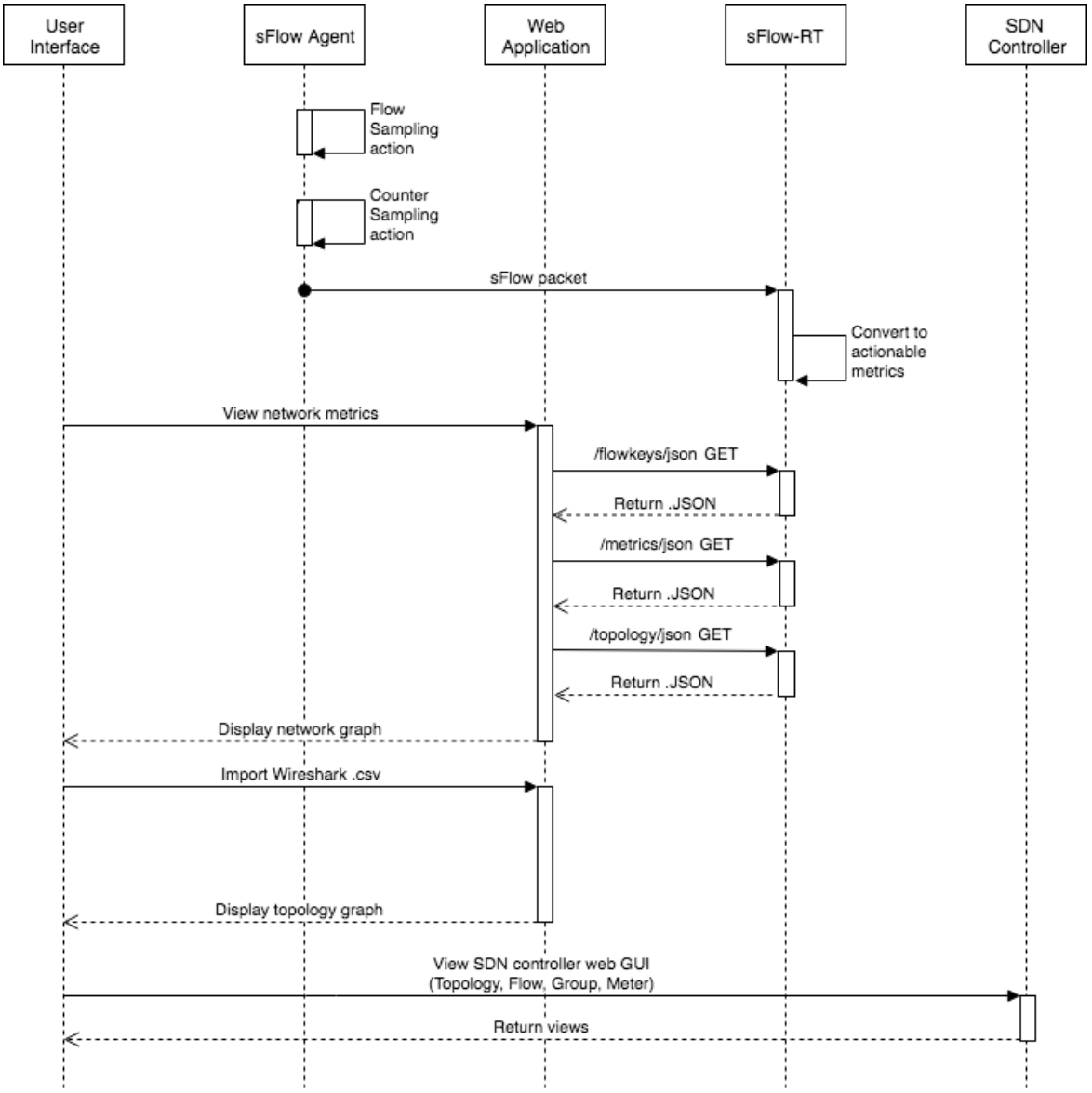

Figure 4.2: Sequence diagram for retrieving and displaying network flows

When comparing the use case diagram, the activity diagram and context diagram it can be deduced that there are two key component flows. There are many different classes that are initiated and subsequent methods and functions that are invoked in order implement the intended expected functionality. The extent of the Mininet source code makes it difficult to generate class diagrams that provide some coherent sense of a system overview.

Therefore a series of sequence diagrams have been constructed to provide an overview of the functional flow of the prototype to further elaborate the diagrams presented in section 3.5.1 and section 4.1.1.

As mentioned in sections 1.6, 2.6 and 3.2, being able to elicit metrics concerning the data and control flows across the emulated network can be considered one of the primary criteria in the development network configuration and monitoring solutions such as the proposed prototype. There is a variety of approaches to configuring, retrieving and displaying these flows.

Figure 4.2, seeks to outline the flow behind retrieving and displaying these flows. It does not include the software utilities packaged in Mininet, but these may be employed in the CLI to get command output prints. The proposed prototype takes three different approaches to eliciting and displaying these flows. It employs Wireshark and OpenFlow dissector, SFlow and sFlow-RT, and SDN controller and OpenFlow.

Wireshark may be continuously run in the background. The intention with employing this together with the OpenFlow dissector plug-in, is to capture network-wide OpenFlow control messages, e.g. Flow-Mod, Group-Mod and Meter-Mod. This data may be exported from Wirehark in a variety of formats, where .csv a the format that may easily manipulated by a web application and represented in an HTML table format.

sFlow is compatible with Mininet and OVS and agents may be embedded in switches and end-hosts to perform packet-based flow and time-based counter sampling that can be relayed to the sFlow-RT analytics engine, in so that the metrics may be queried through the REST API by the companion web application or any SDN application in the application layer.

Any of the selected SDN controllers can employ OpenFlow READ STATE messages to elicit statistical data on flows, groups, meters, etc. and expose these via their respective Northbound REST APIs. Together with sFlow-RT, these APIs may be employed in the development of SDN application over the container-emulated data layer in the prototype testbed, e.g. load balancing.

Figure 4.2 seeks to provide an overview of the conceptual design for creating, starting and displaying a network topology. Illustrated is a series of methods and functions that are invoked, in order to provision distributed Docker container, emulated networking nodes, end-hosts and links, needed to interconnect an emulated data layer in the prototype testbed. The custom topology template script for implementing this is presented in section 5.3.4.

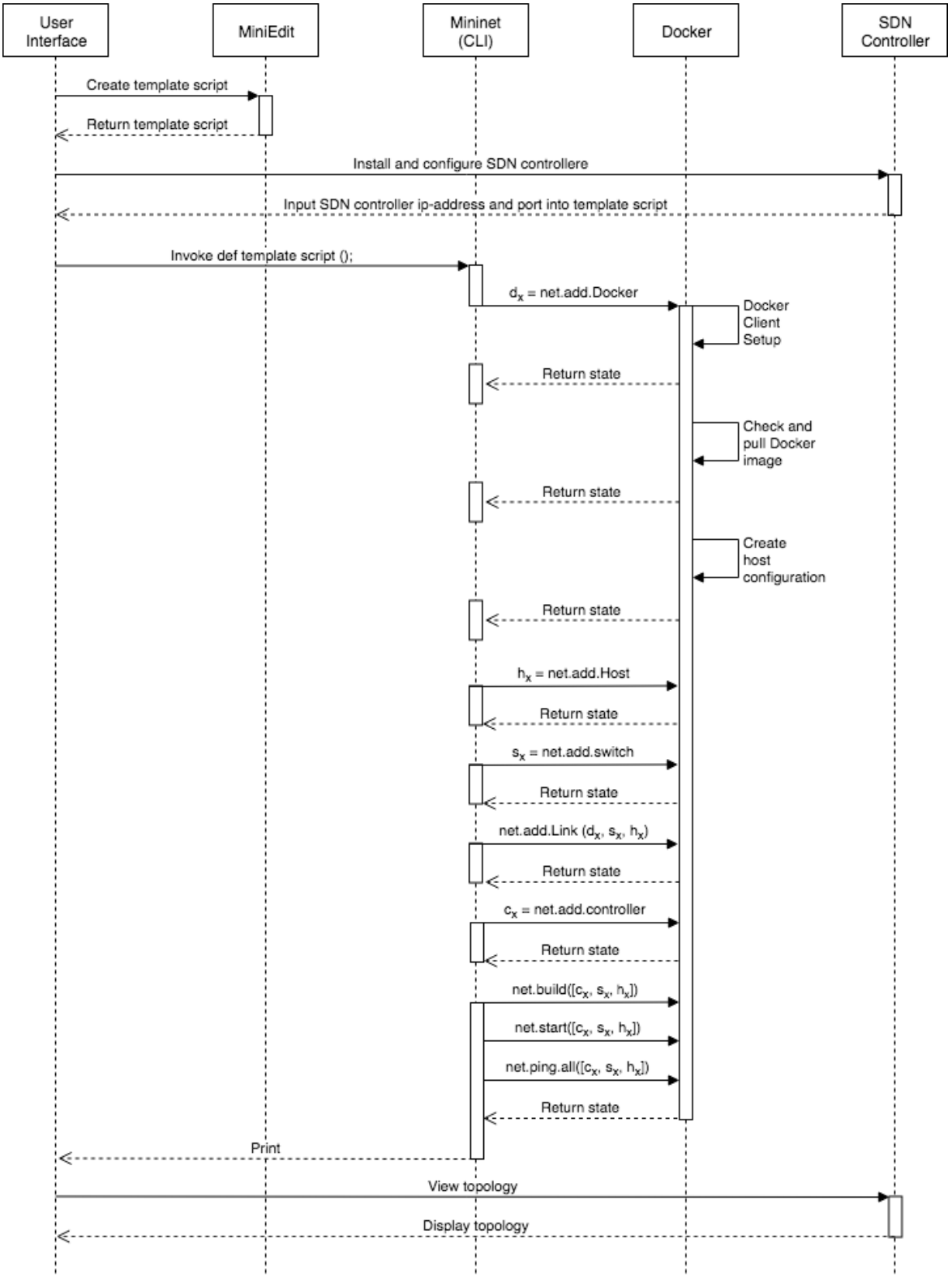

Figure 4.3: Sequence diagram for creating, starting and viewing topology configurations

Chapter 5

Implementation

Chapter 4 sought to describe the conceptual design behind the proposed virtual testbed prototype application. This chapter seeks to describe the implementation process behind this and highlight functional blocks of code that enable the creation of a low-cost emulated testbed that can be dynamically provisioned and scaled to test SDN controller and OpenFlow performance and interaction. During this thesis, the original Mininet API reference architecture was extended and a companion web application constructed, in order to incorporate new features.

5.1 Development Methodology

Together with the project methodological process model, a development methodology was employed to guide the implementation process. This methodological approach is iterative in nature and borrows elements from Extreme programming [121] and SCRUM [36]. The process is detailed in figure 5.1. The argument behind this approach is based on the exploratory nature of the project - continually evolving requirements, fixed deadline and time frame, subjectively and objectively new technologies and implementations.

It builds on continuous development in two-to-three week sprints, which each consists of five distinct phases: research related work and technologies, analyze feasible approaches for implementation, outline functionality and conceptual design, implement required functionality and test functionality. Potential discrepancies may be carried over to the next sprint

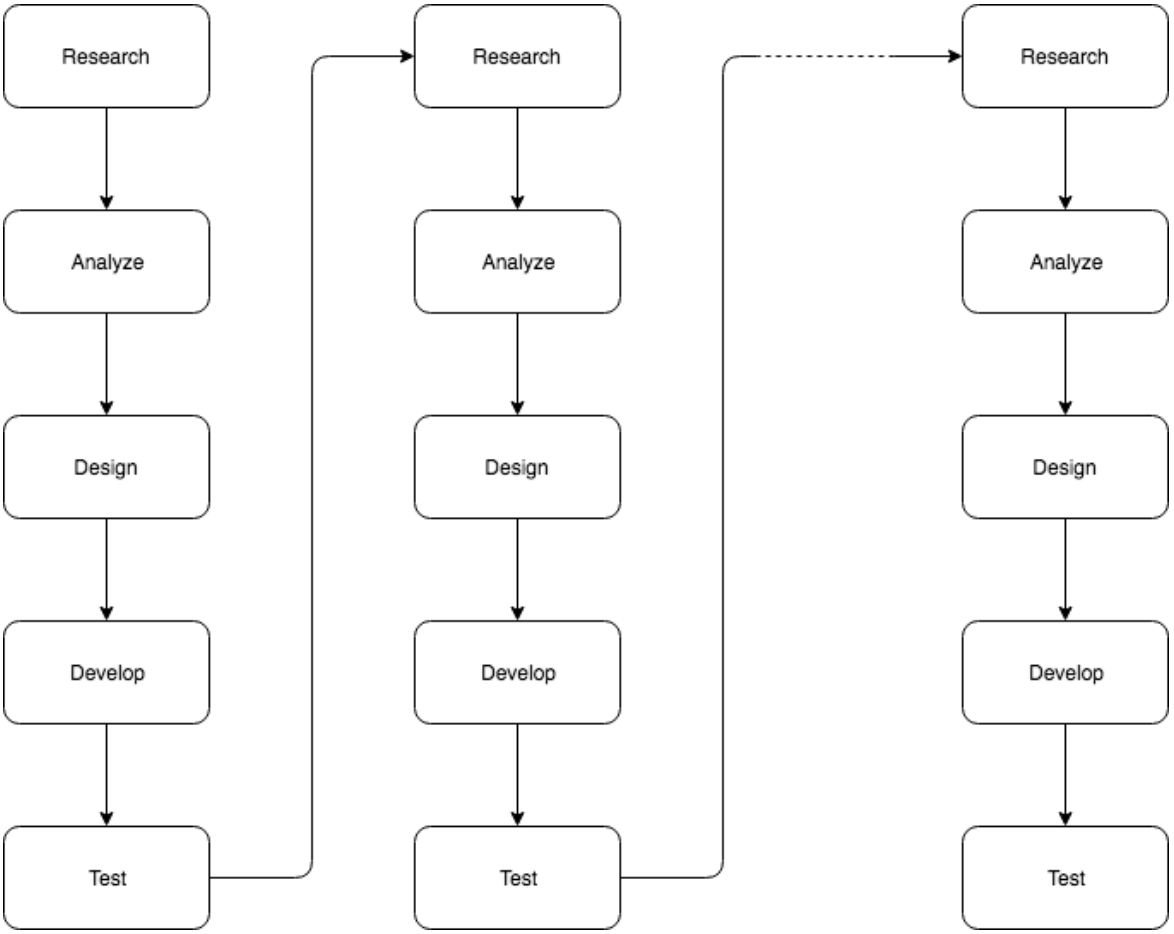

Figure 5.1: Agile development methodology

5.2 Architecture

As mentioned, the thesis project seeks to construct a distributed SDN emulated testbed with a focus on implementing component features that are associated with the control layer and the data layer, as well as establishing interconnection between these and the application layer, so that they may be employed by a given user.

In regards to emulating the data layer, the prototype employs Docker containers to represent Mininet host, the process behind this is described in section 5.3.1. Where the basic division of labor is that networking node (switches) processes execute commands, interfaces (basic Iintf or bandwidth limited TCIntf **FR9**) are assigned and configured and links connect networking nodes together, typically through using Veth pairs (may also tunneled links) **FR4a**. These are handled using Mininet classes and custom sub-classes. sFlow instrumentation is employed within the emulated environment to sample flows and counters that may be relayed to the sFlow Analytics Engine and exposed through its REST

API.

The focus in control layer is predominantly centered on the SDN controller that user has installed and configured and wishes to test and monitor the SDN controller and OpenFlow functionality and traffic through the Southbound API. Here, the topology and control and data network metrics may be visualized by the SDN controllers' GUI, third party software, e.g. Gephi or the companion web application via Wireshark exports and sFlow-RT. As mentioned, the OpenFlow v1.3 is currently the supported version that may be employed **NFR4**.

In reference to figure 1.1, 1.2 and 2.1, the system architecture illustrated in figure 5.1 describes how the different components of the proposed prototype fit together across the different layers.

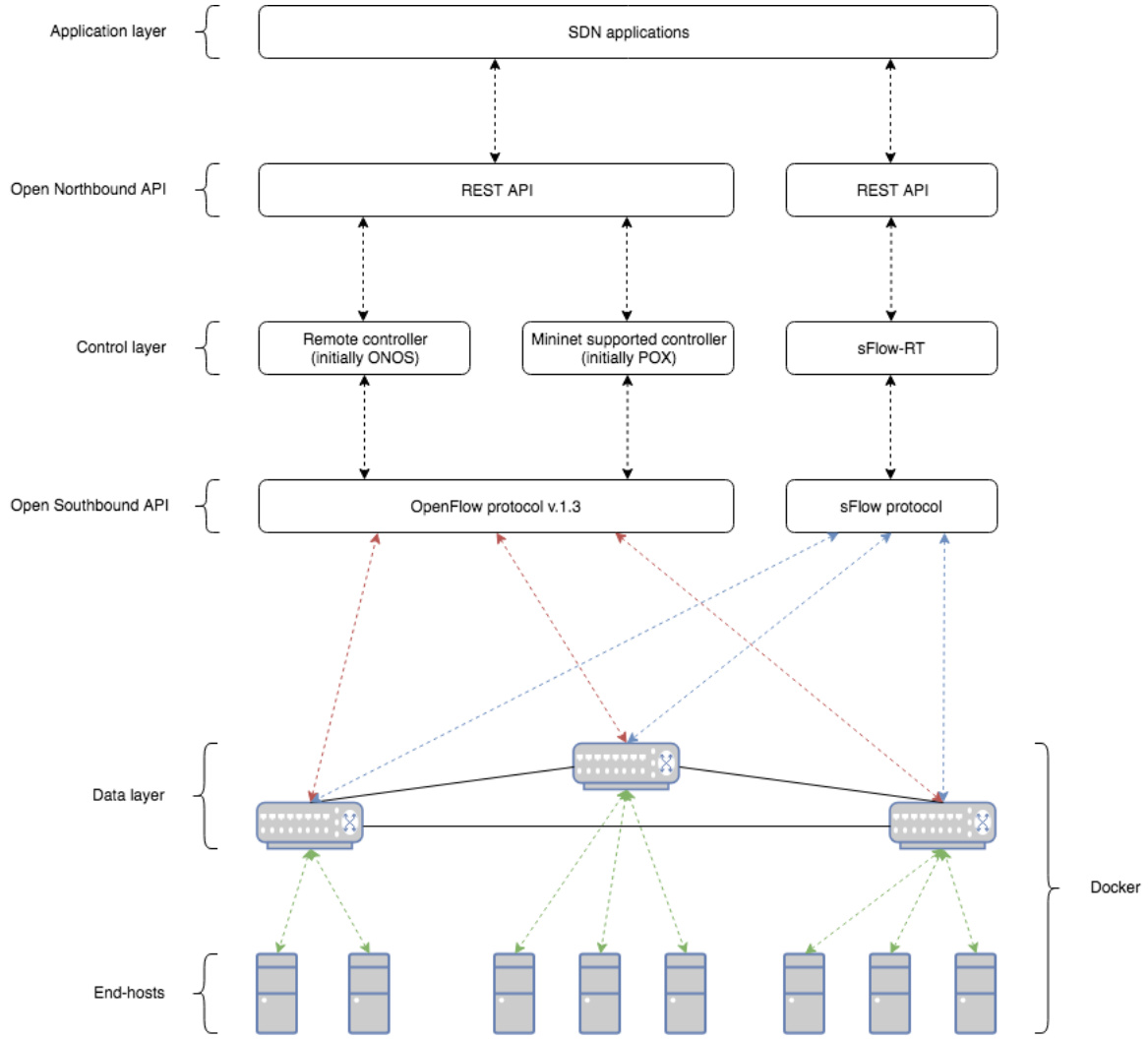

Figure 5.2: System architecture

The intention with the application layer is to expose the SDN controller Northbound REST API and sFlow Analytics Engine REST API or JavaScript methods, so that developer may develop and test SDN application functionality in a development environment before it is implemented in a production setting **FR15**. In relation to section 1.2 and 3.3, this is envisioned to facilitate the migration of OSSs to SDN-capable networks. The prototype may be run on a Linux OS, e.g. IaaS instance, virtual machine or local host [46]**FR12**. IaaS instances have the added benefit that computational resources may be scaled as CPU and RAM requirements increase for larger network topologies.

5.3 Sprints

As delineated in section 5.1, the development of the proposed proof of concept prototype was carried out over a series of sprints. The focus of each sprint was centered around a set of must functional requirements that could be aggregated into a coherent core feature. This section seeks to presents some of the code implementation highlights during these, each with their associated requirement ID references.

5.3.1 Docker Container Provisioning and Mininet Integration

As previously stated, the proposed emulated testbed seeks to employ Docker containers to represent Mininet hosts needed to emulate networking nodes within a given topology. Before a Docker container can be employed as a Mininet hosts, there is a series of configuration steps that has to be taken, as mentioned on section 4.2, in order to provide this functionality. The code examples seek to present some of the programmatic logic behind this process.

The approach to setting up Docker containers is to use Docker instructions, Docker CMD and ENTRYPOINT from Dockerfile to configure the instructions employed to provision a Docker container. Using the Docker.py Python library [118], it is possible to execute these from within a Python application **NFR1**. Allowing the emulated testbed to deploy and manage Docker container, as well as expose container configurations, resources, bindings, mappings and fields to the Mininet API **FR1**. By exposing container resources to the Mininet API, it is possible to impose CPU and memory resource limitations and visualize concurrent resource usage - latter is discussed in section 5.2.7. Together with the variable link parameters (limits to bandwidth, delay, loss and max queue length) and performance modelling features of Mininet, it is possible to create custom network scenarios [89][87] **FR9**.

Figure 5.2 details part of Docker class and associated functions and methods, which is used deploy and configure Docker containers. The initialization (contructor) function is employed to dynamically assign value to object properties, such as **kwarqs, self, name, dockerimage and Dockercmd. Here the objects are separate instances, which each can be employed to execute the required functionality that has been defined within the class,

i.e. perform the series of functions required for configuring a Docker container. The self parameter is a reference pointer to instances within the class.

```
self.volumes = defaults['volumes']
self.environment = {} if defaults['environment'] is None else defaults['environment']
self.network_mode = defaults['network_mode']
self.publish_all_ports = defaults['publish_all_ports']
self.port_bindings = defaults['port_bindings']
self.dns = defaults['dns']
self.dockercli = docker.from_env().api
self._check_image(dockerimage, True)

hc = self.dockercli.create_host_config(
    network_mode=self.network_mode,
    privileged=True,
    binds=self.volumes,
    publish_all_ports=self.publish_all_ports,
    port_bindings=self.port_bindings,
    mem_limit=self.resources.get('mem_limit'),
    cpuset_cpus=self.resources.get('cpuset_cpus'),
    dns=self.dns,
)
```

Figure 5.3: Instances and host configuration

By employing the Docker CLI, the Docker client can be set up. It is a required step before further commands can be issued **NFR7**. The next step is checking for an image that is to be employed to run the Docker container. This image is required to implement Mininet emulations over. The image may be pulled from Dockerhub or locally using tags. However, before a Docker container can be created, a host configuration and resources needs to be specified. Network mode will be set to None, as it will be configured for Mininet network setup, hence why the privileged mode is set to true. Binds/volumes are required mechanisms for mounting persisting data generated by and used by Docker containers through mount points. Furthermore, it is necessary to employ port bindings to publish all the ports the Dockerfile exposes and further supply each container with a prefixed hostname and DNS configuration, in order to connect the containers **FR1**, **FR2**.

In python, default parameter values are evaluated when a function definition is executed, meaning that expressions are evaluated when the function is called. Figure 5.3 details that the Docker class function parameters associated with the creation of Docker containers may have default values. Mutable objects such as lists [] and dicts will be appended new elements, e.g. when creating host configurations and containers, with the inherent default being modified as a result, which makes sense for values such as port bindings, ports and DNS. For some values this will not work and as a result the keyword 'None' is employed. These can values can be saved in the dockerfile and the keyword arguments allows these default values to be modified at run time, which an important part for adding, updating and deleting networking nodes **FR2**.

All container resources are stored in a dictionary so they can be accessed through the Mininet API later. These resources are employed for setting resources limitations and checking container states.

```
defaults = { 'cpu_quota': -1,
             'cpu_period': None,
             'cpu_shares': None,
             'cpuset_cpus': None,
             'mem_limit': None,
             'memswap_limit': None,
             'environment': {},
             'volumes': [],
             'network_mode': None,
             'publish_all_ports': True,
             'port_bindings': {},
             'ports': [],
             'dns': [],
             }
defaults.update( kwargs )

self.resources = dict(
    cpu_quota=defaults['cpu_quota'],
    cpu_period=defaults['cpu_period'],
    cpu_shares=defaults['cpu_shares'],
    cpuset_cpus=defaults['cpuset_cpus'],
    mem_limit=defaults['mem_limit'],
    memswap_limit=defaults['memswap_limit']
)
```

Figure 5.4: Container default values, dictionaries and lists

Figure 5.4 details process of creating and starting a container. The hostname and name prefixes for the container needs to be defined so that the containers can be logically separated. The image needs to be pulled from the repository, so that actions may be performed on the CMD and ENTRYPOINT instruction command fields. This is achieved by inspecting the image. The CMD and ENTRYPOINT fields are defined as lists, so that these may be modified, i.e. overridden. CMD is employed to define default arguments that may be passed to an ENTRYPOINT when a container is created. In this case the container is treated as an executable file. CMD is also employed for issuing ad-hoc commands in a given container [13]. Both CMD and ENTRYPOINT are core tool in defining and running the emulated testbed as a multi-container Docker application **FR1**, **FR2**.

Further, Docker enables the storage of configuration settings and external resource addresses in environment variables. Environment variables are variables that are exposed to processes running inside of Docker containers. These can be set in the dockerfile and modified at run time [13] **FR2**. This assists in realizing multi-container docker environment setups.

Docker supports much of the same functionality as Mininet, and utilize Process Identifier (PID) namespaces, control groups (cgroups) and Linux kernel resource sharing, etc..

By default, all Docker containers have the PID namespace enabled. [13]. As illustrated on figure 3.1, the separation of system process and namespaces hierarchies allow Mininet to provide distinct subnets and isolated network views for the Mininet hosts - an approach that can be mirrored when employed Docker containers as Mininet hosts **NFR3**.

```
self.dockercontainer = self.dockercli.create_container(
            name="%s.%s" % (self.dockernameprefix, name),
            image=self.dockerimage,
            command=self.dockercmd,
            entrypoint=list(),
            stdin_open=True,
            tty=True,
            environment=self.environment,
            host_config=hc,
            ports=defaults['ports'],
            volumes=[self._get_volume_mount_name(v)],
            hostname=name
        )

        self.dockercli.start(self.dockercontainer)
        self.dockercontainerinfo = self.dockercli.inspect_container(self.dockercontainer)
        self.dockerid = self.dockercontainerinfo.get("Id")

        Host.__init__(self, name, **kwargs)
        self.update_resources(**self.resources)
```

Figure 5.5: Container creation, inspection and start up

The Docker API is primarily based on REST, but the HTTP connection can be used to transport input/output (I/O) standard streams, i.e. (stdin) and output (stdout). These I/O streams are opened for shell commands when they are run to enable input and output functionality for the Docker CLI **NFR7**. Furthermore, working with Linux, to handle terminal access the pseudo-tty (PTY) allocated to the docker container needs to be enabled. Technically this emulates a master-slave relationship between a terminal process and text terminal.

By employing standard streams and PTY it is possible to interact with the container process in a session by using Unix pipes - UNIX shell filename piping functionality. In regards to Mininet, the UNIX shell is employed both as a command and scripting language to control shell processes **FR4b**, **NFR6**. This is facilitated by the use of the Python subprocess module, mainly its constituent popen class function and associated object instances. On success, popen returns a pointer to an open stream that can be used to read or write to a specified pipe [122][120]. These provide the necessary support for chaining shell processes in the Linux network namespace and cgroups. These elements allows the system to execute commands inside Docker containers by using the Mininet CLI [13][118] **FR4b**, **NFR3**, **NFR7**.

5.3.2 Virtual Switch Implementation

While the Docker container integration was achieved by subclassing the Mininet Node and Host classes. The Switch superclass and the OVS kernel switch subclass function as they do in Mininet. As mentioned in section 3.1.4, Mininet supports the Stanford Userspace switch, with BOFUSS being an upgraded version of said switch. The BOFUSS git repository provides a bash shell script for installing BOFUSS code components on top of the Userspace switch class. This partially solves a significant issue with the Userspace switch, in that it did not allow datapath operation to enable QoS slicing. As a result, this feature is disabled by default. Implementing the switch subclass as a leaf node and changing a dpopts=" string literal in the extended BOFUSS class enables this feature, but it is a patchwork solution, as it and Mininet are conflict over Traffic Control (TC) queuing at TC interfaces [95][103]. In relation to datapaths, a SDN controller may issue a series of modify-state and read-state messages to alter these over multiple switches.

In Mininet, both the OVS Kernel and the Userspace switches employ dpctl as a management utility tool for querying and configuring switch parameters, e.g. tables and ports. In extension of section 2.1.1, it may be used to send queries for statistics, description, feature and modification of group, meter and flow tables, and ports. In short summary, creating, modifying and deleting datapaths. In regards to the OVS kernel switch, Mininet supports ovs-vsctl and ovs-ofctl for querying and updating the configuration of ovs-vswitchd via the OVSDB protocol and server, and controlling OpenFlow switches and controllers respectively [119][89]. Concurrently, Mininet works best with userspace datapaths [89] **NFR5**.

5.3.3 SDN Controller implementation

As discussed in section 3.1.5, the two chosen controllers that have been decided to work with for the initial implementation and testing, are the POX and ONOS controllers, as one is supported by Mininet and one is a remote controller. This requires varied techniques for controller setup and implementing these into an emulated topology. For SDN controllers to integrate with Mininet, it needs to know the IP-address and Port, to enable connectivity and OpenFlow capabilities **FR8**.

Remote controller Setup

As mentioned in section 2.3 there is a multitude of SDN controllers that can be employed. For the proof of concept prototype, The ONOS controller project has employed for initial tests, because the project has developed a Python module (ONOS.py) [123] that can be employed to create a complete ONOS network by using the Mininet network emulator. Because this integrates with Mininet and is written in Python, it is possible to import classes and sub-classes and use these via the Mininet API. This allows for relatively easy setup and testing. The module introduce does however introduce new prefixes for certain classes

to specialize arguments [106], but by defining default literals command incompatibilities may be circumvented.

Unfortunately, it has not been possible to define SDN controllers within the scope of the node controller class and as such these can concurrently not be represented by a Docker container.

Controller Sub-classing

If the Mininet() constructor is invoked in a script without specifying a controller class, by default it will use the Controller() class to create an instance of the OpenFlow reference controller [46]. However, it is possible to create a controller subclass and pass it to Mininet. For example, a user-defined subclass, i.e. mininet.controller.POX() can invoke a set of modules passed in as options. Figure 5.6 presents an example of how a custom POX controller subclass may be structured.

```
from mininet.node import Controller
from os import environ

POXDIR = environ[ 'HOME' ] + '/pox'

class POX( Controller ):
    def __init__( self, name, cdir=POXDIR,
                  command='python pox.py',
                  cargs=( 'openflow.of_01 --port=%s '
                          'forwarding.l2_learning' ),
                  **kwargs ):
        Controller.__init__( self, name, cdir=cdir,
                             command=command,
                             cargs=cargs, **kwargs )

controllers={ 'pox': POX }
```

Figure 5.6: Controller sub-classing with POX l2 learning example [89]

5.3.4 Topology Generators and Template Scripts

As discussed in the two prior sections, the switch and controller implementations can be employed together with an emulated network topology consisting of the aforementioned controllers and switches, as well as emulated links and end-hosts.

There are different approaches to creating and provisioning a new topology. As with controller subclassing, it is possible to create custom subclasses to the Topo and Mininet classes and import the created Docker class to have the switches and end-hosts be represented by Docker containers. This approach can be employed to create emulated network representation for structured multi-tree topologies with given depth and fanout **FR7**. The primary drawback to this approach is that the links and any given controller needs to be

added and configured manually once the tree topology has been provisioned. The secondary drawback is that this can only defined network architectures with a given depth and fanout limits its applicability in networking scenarios.

```
class TreeTopo( Topo ):
    def build( self, depth=1, fanout=2 ):
        self.hostNum = 1
        self.switchNum = 1
        self.addTree( depth, fanout )
    def addTree( self, depth, fanout ):
        isSwitch = depth > 0
        if isSwitch:
            docker = self.addSwitch( 's%s' % self.switchNum )
            self.switchNum += 1
            for _ in range( fanout ):
                child = self.addTree( depth - 1, fanout )
                self.addLink( docker, child )
        else:
            docker = self.addHost( 'h%s' % self.hostNum )
            self.hostNum += 1
        return docker

def TreeNet( depth=1, fanout=2, **kwargs ):
    topo = TreeTopo( depth, fanout )
    return Mininet( topo, **kwargs )
```

Figure 5.7: Customized tree topology subclass [89]

Another approach is employing the MiniEdit python module. MiniEdit is a simple network editor for Mininet, which implements a GUI canvas for adding nodes, i.e. controllers, switches and end-hosts, and interfaces and links for these **FR4a**. The canvas setup can be compiled in to a topology template script similar in design to figure 5.8, which can then be provisioned by Mininet **FR7**. In so far, it has not been possible to implement Docker functionality into this process, leading to this having to be manually added to the script. However, it may still be employed to map controllers, switches, end-hosts and any intermediary interfaces and links (with built in configuration menu).

An unfortunate detail with either approach is that all Mininet native networking nodes will be booted in default modes, which entails that some configuration is required if a given user seeks to perform custom switching. Further, remote controllers largely require manual configuration before these can be connected with a Mininet topology. For these controllers to be connected in the script, these do need to be supplied with a pre-defined IP address and port **FR8**. Controller sub-classing may alleviate part of this process. Figure 5.7 presents a custom topology template scripts with a ONOS remote controller, two OVS switches and four end-hosts **FR5**. This script has been tested with the OpenDaylight and Floodlight controllers as well.

```
def showcaseNet():

    net = Dockernet(controller=RemoteController, switch=OVSKernelSwitch)

    c1 = net.addController('c1', controller=RemoteController, ip="127.0.0.1", port=6633)

    d1 = net.addDocker('d1', ip='10.0.0.251', dockerimage="ubuntu:xenial")
    d2 = net.addDocker('d2', ip='10.0.0.252', dockerimage="ubuntu:xenial")

    net.addLink(d1, s1)
    net.addLink(s1, s2, cls=TCLink, delay='100ms', bw=1)
    net.addLink(s2, d2)

    s1 = net.addSwitch( 's1' )
    s2 = net.addSwitch( 's2' )

    net.addLink(s1, h1)
    net.addLink(s1, h1)
    net.addLink(s1, h1)
    net.addLink(s2, h2)
    net.addLink(s2, h2)

    h1 = net.addHost( 'h1', ip='10.0.0.1' )
    h2 = net.addHost( 'h2', ip='10.0.0.2' )
    h3 = net.addHost( 'h3', ip='10.0.0.3' )
    h4 = net.addHost( 'h4', ip='10.0.0.4' )

    net.build()
    c1.start()
    c2.start()
    s1.start([c1,c2])
    s2.start([c1,c2])

    net.start()
    net.ping([d1, d2])
    net.staticArp()
    CLI( net )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    showcaseNet()


        c1 = net.addController('c1', controller=RemoteController, ip="192.168.200.22", port=6633)
```

Figure 5.8: Mininet function block script for creating a topology

5.3.5 Network Visualization

As mentioned, network visualization is an important asset to an emulated testbed, in that it aids in creating an overview over the provisioned network. As mentioned in the previous Section, the MiniEdit GUI can assist in this assist in this before a topology is booted, but preferably the implemented SDN controllers should be employed in eliciting topology views once a topology has been emulated.

Given the emulated switches, links and end-hosts, an emulated network topology may

be discovered by a SDN controller through a variety of means, e.g. OpenFlow for configured switches, Address Resolution Protocol (ARP) and Dynamic Host Configuration Protocol (DHCP). Per section 2.6.2, depending on the protocol used to establish this connection, further information, such as device capability and number of ports, may be conveyed to the controller. For any controller to see the end-hosts in Mininet, it is necessary to perform a ping-all command or conduct a ping regression test.

- **SDN controllers with web-based GUI** - As displayed in figure 2.9, ONOS, OpenDaylight, Beacon and Floodlight have implemented web-based GUI that can be employed to provide a topological view of the network **FR10**.
- **SDN controllers that require third party software** - Other SDN controllers, NOX, POX and RYU, requires that a third party software tool, such as Gephi [124], be employed to visualize the topological view **FR10**.

Because sFlow agents in the emulated switches can relay information in sFlow datagrams and packets concerning the interface index, ipsource, ipdestination, macsource, macdestination and packet-based flows and time-based counters associated with interface and links, these may be queried from the sFlow-RT REST APIs **FR6**. This information allows the companion web application, using the vis.js library [125], to display dynamic generated and automatically organized network views. These network views may display the switches and links in relation to the flow size. Alternatively the topological views from Gephi may be parsed using the vis.js library **FR10**.

An alternative approach is employing a third party solution called Narmox [126]. Narmox can construct a topology view by importing print outputs from mininet>dump and mininet>links commands. This does however require manual input and it does not dynamically update when the topology changes. Nor does it include any metrics.

5.3.6 Network monitoring and Data Representation

As mentioned in section 3.2, there are multiple ways of eliciting network statistics and resource usage from an emulated network topology, in order to evaluate the interaction between SDN controllers, OpenFlow and networking nodes.

Mininet provides a suite of functionality, such as checking flow tables and port statistics using dptcl commands, and tools for checking bandwidth (Bandwidth Monitor NG, ethstats), latency (ping), queues (TC), TCP Congestion Window statistics (TCP probe), CPU usage (global / CPU Accounting (CPUAcct)) [46].

Where most SDN controllers, including POX and ONOS, can provide flow, port, group, meter and queue statistic services about the network they are controlling. The primary approach to eliciting these throughout the project has been to employ cURL to test SDN controller REST API endpoints, but the output may be viewed in SDN controllers with web-GUIs as well.

As presented in section 2.7.1, there is variety of network monitoring protocols that can assist in providing statistic services for a given emulated network. As delineated in section 3.2, sFlow may be and has been employed to monitor and analyze the network flow and counters. A companion web application written in JavaScript, HTML and CSS, has been developed with the purpose of providing chart capabilities trough the use of the Vis.js library and sFlow-RT REST API **FR6**, **NFR2**. Figure 5.9 showcase the flowcount function, which provide a subset of the flow and counter calculations and an interval handler needed to dynamically update Vis charts. This enables a user to visually monitor data network traffic in a GUI.

```
function flowCount(flow) {
  var respond = activeFlows('topology',flow,1,0);
  retur respond && respond.length > 0 ? respond[0].value : 0;
}

setInterval(function(now) {
  points = {};
  points['diameter'] = topologyDiameter();
  var bytespersecond = flowCount('mn_bytes') * 8;
  points['flows'] = calculateFlows('mn_flow',1,points.bytespersecond);
  points['interfaces'] = calculateInterfaces('mn_bytes');

  trend.addPoints(now,points);
},1);
```

Figure 5.9: Subset of flow and counter calculations

Concurrently, it is possible to check two different overview modes. These are port interfaces statistics in bytes per second and flow statistics in bytes per second. The intend is to extend this to include a visual overview of the service statistics provided by SDN controllers. This has been prioritized lower, as some controllers already provides a graphical overview using the same REST API calls.

Mininet end-hosts can run any command or application that is available to the underlying system, as a result thereof, it is possible to host an iPerf server on one end-host and an iPerf client on another to test the available capacity between the two end-host [46]. iPerf and ping regression tests have been employed to provide data for the sFlow-RT web application **FR9**.

The basic packet capture and extended OpenFlow Wireshark dissector tool has been employed extensively throughout the project to check the OpenFlow protocol and control traffic. These captured packets may be exported in a Comma Separate Value (.csv) file, which can be manipulated using an AJAX jQuery method and HTML scripting on the client-side, so that it may be displayed in a HTML table **FR6**.

Chapter 6

Discussion

The goal of this project was to perform a study on the development of SDN technologies and how they are and can be applied in CSP networks and to further see how this may be implemented and function in an emulated testbed. How such a testbed may function and how the architecture may be implemented has been the primary objective of the thesis. However, due to time and resource constraints, certain limitations had to be imposed on the project. As a result thereof, this section seeks to evaluate the process and prototype.

6.1 Agile process approach

The project methodology is described in sections 1.7.2 and 5.1 respectively. These delineate the primary approach to writing the report and implementing the prototype, and served as an overall overview and guide. In hindsight, an agile methodological approach to investigating the constituent fields of the report and implementing associated features, functioned well. This conclusion is made on the basis that it enabled greater degrees of flexibility in relation to implementing and changing features or researching and updating chapters when new observations were made. It is further made in regards to, how the selected topic proved to be substantial and required an extensive period of time from the allotted time-frame, to be dedicated to research into the domain fields.

As a result thereof, a larger amount of time was employed gathering domain knowledge than was originally intended, in order to understand the premises behind these fields and their associated solutions, before actual development and implementation could occur. In order to alleviate this impact, efforts should be made to approach domain knowledge gathering and analysis from a more structured approach, with shorter intervals between meetings to facilitate research direction and efforts. Overall, despite the described fluctuations, the selection of process models may be deemed successful, albeit with room for improvement regarding the preliminary research phase.

6.2 Prototype approach

The prototype presented in chapter 4 and 5, is technically a complete system within itself, based solely on open source elements. As mentioned, the purpose of the emulated testbed prototype is to allow developers to gain insight into SDN controller and OpenFlow interactions with the underlying data layer. These interactions are enabled through the emulation of virtual switches and end-hosts represented by isolated Docker containers, which may interact with the deployed SDN controller and sFlow network monitoring protocol. With regards to the application layer, this open source approach should enable developers to construct SDN applications and test functionality in the emulated development environment before this is deployed in a production setting.

Because everything is based on open source solutions, technologies and protocols and the prototype is written in extensible Python (emulation testbed) and JavaScript (companion web application), this enables any developer to further introduce or tweak presented functionality to accommodate for more complex use cases, such as integrating other network configuration and monitoring protocols or legacy switches and routers.

The open source nature of the proposed prototype does however impose certain limitations in regards to which proprietary solutions, technologies and protocols that may be concurrently deployed together with the emulated testbed without requiring privileged access to the source code of these elements. As a case, Motorola's desire to migrate to Juniper's Contrail controller. Further, because it is open source, it does not support the entire super set of the wide range of virtual network functions that are concurrently available, e.g. UML switch.

One of the limitations in regards to presented functionality is that the proposed prototype does not integrated with OpenStack and orchestration platforms, e.g. Kubernetes, OpenShift, CloudFoundry and Mesos. A result of Mininet not being able to directly integrate with respective API end-points. However, it should be noted that this may be alleviate through python modules that may translate input and output between the emulated testbed and the mentioned solutions to present API-like interfaces.

Another limitation in regards to the presented functionality is that the companion web application does not have the functionality to query the separate REST APIs associated with the various SDN controllers that may be integrated with the emulated testbed, simply as a result of the REST APIs having differentiated URLs and commands. An approach to solving in the immediate future would be to implement extensive switch statements that can be used to perform different actions and requests based on changing conditions, i.e use switch statements to select and execute code blocks. This may have fallback functionality through the use of the default keyword and code block.

A further limitation is that the emulated testbed, in extension of it being based upon the network emulator framework of Mininet, concurrently can only support OpenFlow v1.3. While this version has most of the core functionality that may be employed with OpenFlow, it does not enable synchronized tables and (scheduled) bundles for configuring

multiple switches at the same time and it does not include egress table functionality.

The employment of containerization in the emulated testbed does come with certain disadvantages when up-scaling network represent large fat tree topologies. There can be a distinct loss of accuracy with regards to network traffic capacity and flow metrics, when conducting performance modelling over such a scale. As networking node density is increased substantially, the pressure on random access memory can result in adverse and irregular effects during high node count scenario experiments. This is an issue that is recognized by Mininet. However, for the the vast majority of deployment scenarios, there will be minuscule impact to be observed.

In regards to Southbound API protocols, the primary focus has been on implementing and monitoring OpenFlow operations within the emulated testbed. As illustrated in figure 2.9 and further described in section 2.6.2, some SDN controllers, such as ONOS and OpenDaylight support a wider array of network configuration protocols that they may employ to configure the underlying switches. This become increasingly relevant when deploying scenarios that consists of OpenFlow-capable and legacy switches in one network topology. In hindsight, more effort should have gone towards testing the operations behind these.

As mentioned in section 5.3.4, template generation tools, such as MiniEdit.py does not concurrently support the Docker functionality that has been implemented in node classes of Mininet. It therefore requires manual setup in the compiled script. A large step towards facilitating the deployment process would be to ensure that the node classes in MiniEdit automatically provision a container when they are added to the canvas.

Chapter 7

Conclusion

The instigated challenge for this project was to find an approach for performing network emulation to replicate data layer switch topologies with associated end-hosts, as to conduct performance modelling on the emulated network to test and observe SDN controller and OpenFlow interaction and implementation. An approach that could assist in the facilitation the migration from traditional networking to Software-Defined Networking, by allowing the development of SDN applications in a portable emulated environment. Based on this, primary and secondary research questions were defined.

How can network emulation and monitoring technologies be employed to investigate the interaction between data and control layers components in an emulated SDN architecture?

By employing Docker containers and the Mininet emulator as Linux-based paravirtualization techniques, it is possible to emulate scalable and encapsulated network elements, such as the User- and Kernal-space Open vSwitch, and end-hosts in a network topology. As Mininet hosts can run any command or application that is available to the underlying OS (Linux) and its file system, it is further possible to conduct performance modelling via iperf or regression tests and parametric emulated links. By integrating a SDN controller and the OpenFlow protocol into this emulated network topology with modelled traffic, the interaction between the SDN controller, OpenFlow and emulated switches can be investigated. As discussed, this can be achieved in different ways. The proposed prototype makes use of a companion web application to query, import and display control and data network metrics. These metrics are captured using the sFlow monitoring protocol and the Wireshark OpenFlow dissector, which are both compatible with Mininet.

As stipulated in section 1.4, in order to answer the primary research question, each sub-questions seek to provide insight into the underlying problems behind a potential implementation.

How may network emulation technologies be employed to provision a network topology?

Mininet uses process-based virtualization (network namespaces) to provide individual processes with separate network interfaces, routing and ARP tables, allowing it to emulate networking nodes over a Linux kernel. As discussed, Docker containers may represent Mininet hosts. This enables the emulated testbed built on the Mininet emulator, to provision, scale, limit (CPU, memory, disk I/O, network - using cgroups) and delete isolated containers at run time. This approach effectively, allows the emulated testbed to be employed as small-scale emulated cloud infrastructure, in which computational resources may be delegated. This is achievable because Docker and Mininet employ many of the same Linux kernel features.

As Mininet and the Docker containers are based on Linux kernel processing, these are heavily dependent on the available computational resources to scale to larger topologies. Therefore, hosting the prototype on a IaaS platform, may enable those computational resources to be scaled dynamically, as it may be required by provisioned network topology. It may also be employed on a desktop or laptop with certain computational restrictions.

How can network control and monitoring be implemented in an emulated testbed?

With the emulated data layer in place, SDN controllers can be installed on a local host or IaaS instance, to discover and perform network control and monitoring on the provisioned network using the OpenFlow protocol. Connection between the controller and the emulated testbed is established by pointing to the IP-address and port of the SDN controller, e.g. 127.0.0.1 and 6633. The SDN controller may employed a combination of OpenFlow messages to configure and elicit emulated switches in the network, e.g. read state and modify state messages. The observed data is exposed through the respective controllers' Northbound REST APIs.

As the sFlow protocol may be integrated with Mininet and OVS, it has been employed to substantiate this process, by capturing packet-based flows and time-based counters. This telemetry may be aggregated by the sFlow Analytics engine and converted to actionable metrics that is exposed through its REST APIs in a JSON format. These APIs can be queried by a companion web application, to retrieve and display data flows in graphs.

To further substantiate these processes, the Wireshark packet capture software tool and the OpenFlow dissector plug-in, has been employed to capture, filter and inspect packets. While this can be used to sniff both data and control traffic, it has mainly been employed to filter for the OpenFlow protocol.

How may network control and data flow metrics be displayed to the user?

The integrated SDN controllers that have a web-GUI for visualizing REST API output, e.g. Flow, Group, Meter and Port, enables statistics to be view directly. Alternatively, cURL or the companion web application may be employed to represent the output. This is more predominantly employed for NOX, POX and RYU controllers that require third party software to visualize REST API outputs.

The sFlow protocol and sFlow-RT Analytics Engine may be employed to relay and convert flows and counters to a JSON format that may be retrieved via the sFlow REST API. These can can be rendered in graphs or charts by the companion web application using the Vis.js browser-based visualization library and HTML scripts. Furthermore, as Wireshark may export captured packets in a .csv format, the companion web application may convert these to a HTML/CSS table using client-side methods.

How can a provisioned network topology be visualized?

Similarly to how statistics services may be visualized in SDN-controller web-GUIs, so may the discovered topology be visualized by employing SDN controller topology views. As sFlow agents is embedded in the network topology and sFlow-RT receives Interface Index, ipsource, ipdestination, macsource, macdestination and packet-based flows and time-based counters, it is possible to visualize the links between the switches using the aforementioned Vis.js library and HTML scripts.

Bibliography

[1] *5G-PPP - Vision on Software Networks and 5G*. https://5g-ppp.eu/wp-content/uploads/2014/02/5G-PPP_SoftNets_WG_whitepaper_v20.pdf. (Accessed on 03/01/2019). 2017.

[2] Keshav Sood, Shui Yu, and Yong Xiang. "Software-defined wireless networking opportunities and challenges for Internet-of-Things: A review". In: *IEEE Internet of Things Journal* 3.4 (2016), pp. 453–463.

[3] Wenfeng Xia et al. "A survey on software-defined networking". In: *IEEE Communications Surveys & Tutorials* 17.1 (2015), pp. 27–51.

[4] Mao Yang et al. "Software-defined and virtualized future mobile and wireless networks: A survey". In: *Mobile Networks and Applications* 20.1 (2015), pp. 4–18.

[5] *Cisco Visual Networking Index: Forecast and Trends, 2017–2022*. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.pdf. (Accessed on 02/19/2019). 2018.

[6] Giorgos Papastergiou et al. "De-ossifying the internet transport layer: A survey and future perspectives". In: *IEEE Communications Surveys & Tutorials* 19.1 (2017), pp. 619–639.

[7] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The road to SDN: an intellectual history of programmable networks". In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 87–98.

[8] Bruno Astuto A Nunes et al. "A survey of software-defined networking: Past, present, and future of programmable networks". In: *IEEE Communications Surveys & Tutorials* 16.3 (2014), pp. 1617–1634.

[9] Akram Hakiri et al. "Software-defined networking: Challenges and research opportunities for future internet". In: *Computer Networks* 75 (2014), pp. 453–471.

[10] Y. Yuhong Q. Duan and V. Athanasios. "A survey on service-oriented network virtualization toward convergence of networking and cloud computing". In: *IEEE Transactions on Network and Service Management* 9.4 (2012), pp. 373–392.

[11] *Kubernetes Documentation - Kubernetes*. https://kubernetes.io/docs/home/. (Accessed on 04/04/2019).

[12] *Home | Red Hat OpenShift Documentation*. https://docs.openshift.com/. (Accessed on 04/04/2019).

[13] *Docker Documentation | Docker Documentation*. https://docs.docker.com/. (Accessed on 04/04/2019).

[14] *OpenStack Docs: Rocky*. https://docs.openstack.org/rocky/?_ga=2.12862581.356515881.1554585408-1837668940.1548626759. (Accessed on 04/06/2019).

[15] Mark Richards. *Microservices vs. service-oriented architecture*. O'Reilly Media, 2015.

[16] *The programmable network cloud – enriching the cloud with NFV and SDN*. https://www.ericsson.com/assets/local/news/2016/03/wp-the-programmable-network-cloud.pdf. (Accessed on 04/08/2019). 2016.

[17] Mani Prashanth Varma Manthena et al. "An SDN-based Architecture for Network-as-a-Service". In: *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2015, pp. 1–5.

[18] Arsany Basta et al. "A virtual SDN-enabled LTE EPC architecture: A case study for S-/P-gateways functions". In: *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*. IEEE. 2013, pp. 1–7.

[19] *Next-Generation Architecture for Cable Operator Networks: SDN and Network Function Virtualization White Paper*. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/cable-access-solutions/white-paper-c11-732736.pdf. (Accessed on 02/27/2019).

[20] *OpenFlow Switch Specification - Version 1.5.1 (Protocol version 0x06)*. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf. (Accessed on 03/04/2019). 2015.

[21] *Software-Defined Networking (SDN) Definition - Open Networking Foundation*. https://www.opennetworking.org/sdn-definition/. (Accessed on 03/04/2019).

[22] *Cilium*. https://cilium.io/. (Accessed on 06/03/2019).

[23] *SDN and NFV: Transforming the Service Provider Organization*. https://www.juniper.net/assets/us/en/local/pdf/whitepapers/2000579-en.pdf. (Accessed on 02/27/2019).

[24] *Software-Defined Networking (SDN) - Juniper Networks*. https://www.juniper.net/us/en/products-services/sdn/. (Accessed on 02/27/2019).

[25] *ONF - SDN migration considerations and use cases*. https://www.opennetworking.org/wp-content/uploads/2014/10/sb-sdn-migration-use-cases.pdf. (Accessed on 03/01/2019). 2014.

[26] Massimo Fatato Ron Angner Michael Dargue and Shahed Mazumder. "The Future of Networks Report: Dealing with Transformation in a Virtualized World". In: Broadband Forum, 2017, pp. 1–12.

[27] *Next Generation Mobile Alliance - 5G White Paper*. https://www.ngmn.org/fileadmin/ngmn/content/images/news/ngmn_news/NGMN_5G_White_Paper_V1_0.pdf. (Accessed on 03/01/2019).

[28] Emeka Obiodu and Mark Giles. *The 5G era: Age of boundless connectivity and intelligent automation*. https://www.gsmaintelligence.com/research/?file=&download. (Accessed on 03/01/2019). 2017.

[29] European telecommunications Standards Institute. *Network Operator Perspectives on NFV priorities for 5G*. https://portal.etsi.org/nfv/nfv_white_paper_5g.pdf. (Accessed on 03/01/2019). 2017.

[30] *Resources - IEEE Software Defined Networks*. https://sdn.ieee.org/outreach/resources. (Accessed on 03/01/2019).

[31] *ONF - SDN Use Cases and Migration Methods*. https://www.opennetworking.org/images/stories/downloads/sdn-resources/use-cases/Migration-WG-Use-Cases.pdf. (Accessed on 03/03/2019). 2014.

[32] *Apache Mesos - Documentation Home*. http://mesos.apache.org/documentation/latest/. (Accessed on 04/06/2019).

[33] Lindinkosi L Zulu, Kingsley A Ogudo, and Patrice O Umenne. "Simulating Software Defined Networking Using Mininet to Optimize Host Communication in a Realistic Programmable Network". In: *2018 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD)*. IEEE. 2018, pp. 1–6.

[34] Chandan Pal et al. "Implementation of simplified custom topology framework in Mininet". In: *2014 Asia-Pacific Conference on Computer Aided System Engineering (APCASE)*. IEEE. 2014, pp. 48–53.

[35] *PowerPoint Presentation*. https://www.juniper.net/assets/us/en/local/pdf/nxtwork/alliance-perspective-challenges-and-best-practices-for-deploying-nfv-sdn-redhat.pdf. (Accessed on 04/08/2019).

[36] *What is Scrum?* https://www.scrum.org/resources/what-is-scrum. (Accessed on 06/03/2019).

[37] *SDN architecture*. https://www.opennetworking.org/wp-content/uploads/2013/02/TR_SDN_ARCH_1.0_06062014.pdf. (Accessed on 03/04/2019). 2014.

[38] *Deployment Case Study of SDN and NFV Transformation*. https://www.cisco.com/c/dam/global/en_au/assets/ciscolive/pdfs/deployment-case-study-of-sdn-and-nfv-transformation.pdf. (Accessed on 04/09/2019).

[39] Michel S Bonfim, Kelvin L Dias, and Stenio FL Fernandes. "Integrated NFV/SDN architectures: A systematic literature review". In: *arXiv preprint arXiv:1801.01516* (2018).

[40] *What's Software-Defined Networking (SDN)? - SDxCentral*. https://www.sdxcentral.com/networking/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/. (Accessed on 03/04/2019).

[41] *Impact of SDN and NFV on OSS/BSS - ONF Solution Brief*. https://www.opennetworking.org/wp-content/uploads/2014/10/sb-OSS-BSS.pdf. (Accessed on 05/29/2019).

[42] *Uniform Resource Identifier (URI): Generic Syntax*. https://www.ietf.org/rfc/rfc3986.txt. (Accessed on 04/10/2019). 2005.

[43] *Open vSwitch Documentation - Release 2.6.0*. https://media.readthedocs.org/pdf/ovs-reviews/latest/ovs-reviews.pdf. (Accessed on 03/05/2019). 2017.

[44] *GitHub - Openv sSwitch: tracking http://openvswitch.org/*. https://github.com/homework/openvswitch. (Accessed on 03/17/2019).

[45] *OVSDB — Open vSwitch 2.11.90 documentation*. http://docs.openvswitch.org/en/latest/ref/ovsdb.7/. (Accessed on 03/21/2019).

[46] *Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet*. http://mininet.org/. (Accessed on 03/05/2019).

[47] Eder Leao Fernandes et al. "The Road to BOFUSS: The Basic OpenFlow User-space Software Switch". In: *arXiv preprint arXiv:1901.06699* (2019).

[48] OpenDaylight Project. *OpenDaylight Documentation - Release Beryllium*. https://media.readthedocs.org/pdf/opendaylight/stable-beryllium/opendaylight.pdf. (Accessed on 03/16/2019). 2017.

[49] *Documentation « Floodlight OpenFlow ControllerProject Floodlight*. http://www.projectfloodlight.org/documentation/. (Accessed on 03/16/2019).

[50] J. Parrott. *Nox Documentation - Release 0.17.0*. https://media.readthedocs.org/pdf/nox/stable/nox.pdf. (Accessed on 03/16/2019). 2017.

[51] M. McKerns. *Pox Documentation - Release 0.2.6.dev0*. https://media.readthedocs.org/pdf/pox/latest/pox.pdf. (Accessed on 03/16/2019). 2019.

[52] *Quick Start - Beacon - Confluence*. https://openflow.stanford.edu/display/Beacon/Quick+Start.html. (Accessed on 06/06/2019).

[53] Ryu Development Team. *Ryu Documentation - Release 4.30*. https://media.readthedocs.org/pdf/ryu/latest/ryu.pdf. (Accessed on 03/16/2019). 2018.

[54] *ONOS - Development Guide*. https://wiki.onosproject.org/display/ONOS/Guides. (Accessed on 03/16/2019).

[55] *Network Functions Virtualisation - An Introduction, Benefits, Enablers, Challenges and Call for Action*. https://portal.etsi.org/nfv/nfv_white_paper.pdf. (Accessed on 03/04/2019). 2012.

[56] *OpenSDNCore | Virtualisation Testbed for NFV/SDN Environment*. https://www.openSDNcore.org/. (Accessed on 05/04/2019).

[57] *GS NFV-INF 004 - V1.1.1 - Network Functions Virtualisation (NFV); Infrastructure; Hypervisor Domain*. https://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/004/01.01.01_60/gs_nfv-inf004v010101p.pdf. (Accessed on 04/12/2019).

[58] *GS NFV 002 - V1.2.1 - Network Functions Virtualisation (NFV); Architectural Framework*. https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf. (Accessed on 03/06/2019). 2014.

[59] *GS NFV-MAN 001 - V1.1.1 - Network Functions Virtualisation (NFV); Management and Orchestration*. https://www.etsi.org/deliver/etsi_gs/nfv-man/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf. (Accessed on 04/13/2019).

[60] *OSM Public Wiki*. https://osm.etsi.org/wikipub/index.php/Main_Page. (Accessed on 05/31/2019).

[61] *OPNFV Documentation — Hunter documentation*. https://docs.opnfv.org/en/stable-hunter/. (Accessed on 05/31/2019).

[62] *Network Basics: TCP/IP Protocol Suite*. https://www.dummies.com/programming/networking/network-basics-tcpip-protocol-suite/. (Accessed on 06/06/2019).

[63] Network Working Group - IETF. *Requirements for Internet Hosts – Communication Layers*. https://www.ietf.org/rfc/rfc1122.txt. (Accessed on 03/16/2019). 1989.

[64] Network Working Group - IETF. *Requirements for Internet Hosts – Application and Support*. https://www.ietf.org/rfc/rfc1123.txt. (Accessed on 03/16/2019). 1989.

[65] Network Working Group - IETF. *Simple Network Management Protocol (SNMP)*. https://tools.ietf.org/html/rfc1157. (Accessed on 03/16/2019). 1990.

[66] Network Working Group - IETF. *Architecture for SNMP Frameworks*. https://www.ietf.org/rfc/rfc2571.txt. (Accessed on 03/18/2019). 1999.

[67] Network Working Group - IETF. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information*. https://tools.ietf.org/html/rfc5101. (Accessed on 03/16/2019). 2008.

[68] Rick Hofstede et al. "Flow monitoring explained: From packet capture to data analysis with netflow and ipfix". In: *IEEE Communications Surveys & Tutorials* 16.4 (2014), pp. 2037–2064.

[69] Network Working Group - IETF. *InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. https://www.ietf.org/rfc/rfc3176.txt. (Accessed on 03/16/2019). 2001.

[70] *Understanding SPAN,RSPAN,and ERSPAN - Cisco Community*. https://community.cisco.com/t5/networking-documents/understanding-span-rspan-and-erspan/ta-p/3144951. (Accessed on 06/05/2019).

[71] Network Working Group - IETF. *Network Configuration Protocol (NETCONF)*. https://tools.ietf.org/html/rfc6241. (Accessed on 03/16/2019). 2011.

[72] Network Working Group - IETF. *The Open vSwitch Database Management Protocol*. https://tools.ietf.org/html/rfc7047. (Accessed on 03/16/2019). 2013.

[73] *CORE IPC API - verion 1.23*. https://downloads.pf.itd.nrl.navy.mil/docs/core/core_api.pdf. (Accessed on 04/16/2019).

[74] Martin Taylor. *THE APPLICATION OF CLOUD NATIVE DESIGN PRINCIPLES TO NETWORK FUNCTIONS VIRTUALIZATION*. http://info.metaswitch.com/hubfs/PDFs/the-application-of-cloud-native-design-principles-to-network-functions-virtualization-white-paper.pdf. (Accessed on 04/20/2019).

[75] Qi Zhang et al. "A comparative study of containers and virtual machines in big data environment". In: *arXiv preprint arXiv:1807.01842* (2018).

[76] *Are Containers Replacing Virtual Machines? - Docker Blog*. https://blog.docker.com/2018/08/containers-replacing-virtual-machines/. (Accessed on 05/31/2019).

[77] *Cloud Foundry Docs*. https://docs.cloudfoundry.org/. (Accessed on 06/01/2019).

[78] Carl Boettiger. "An introduction to Docker for reproducible research". In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79.

[79] Paolo Di Tommaso et al. "The impact of Docker containers on the performance of genomic pipelines". In: *PeerJ* 3 (2015), e1273.

[80] David Bernstein. "Containers and cloud: From lxc to docker to kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[81] *Apache Spark - Unified Analytics Engine for Big Data*. https://spark.apache.org/. (Accessed on 04/23/2019).

[82] Roberto Morabito. "Power consumption of virtualization technologies: an empirical investigation". In: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2015, pp. 522–527.

[83] Janki Bhimani et al. "Accelerating big data applications using lightweight virtualization framework on enterprise cloud". In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2017, pp. 1–7.

[84] *Docker Bridge Networking Deep Dive – Peng Xiao – Medium*. https://medium.com/@xiaopeng163/docker-bridge-networking-deep-dive-3e2e0549e8a0. (Accessed on 05/14/2019).

[85] Yiming Xu, V Mahendran, and Sridhar Radhakrishnan. "SDN docker: Enabling application auto-docking/undocking in edge switch". In: *2016 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*. IEEE. 2016, pp. 864–869.

[86] *A network testbed for commercial telecommunications product testing*. http://imunes.net/dl/papers/softcom_2014.pdf. (Accessed on 04/24/2019).

[87] *Reproducing Network Research | network systems experiments made accessible, runnable, and reproducible*. https://reproducingnetworkresearch.wordpress.com/. (Accessed on 04/24/2019).

[88] *MININET - Open Networking Foundation*. https://www.opennetworking.org/mininet/. (Accessed on 04/29/2019).

[89] *Introduction to Mininet · mininet/mininet Wiki · GitHub*. https://github.com/mininet/mininet/wiki/Introduction-to-Mininet. (Accessed on 04/25/2019).

[90] *Cluster Edition Prototype - Mininet/mininet Wiki - GitHub*. https://github.com/mininet/mininet/wiki/Cluster-Edition-Prototype. (Accessed on 04/24/2019).

[91] *MaxiNet: Distributed Network Emulation*. https://maxinet.github.io/. (Accessed on 04/24/2019).

[92] Richard Cziva et al. "Container-based network function virtualization for software-defined networks". In: *2015 IEEE symposium on computers and communication (ISCC)*. IEEE. 2015, pp. 415–420.

[93] Richard Cziva, Simon Jouet, and Dimitrios P Pezaros. "Gnfc: Towards network function cloudification". In: *2015 IEEE conference on network function virtualization and software defined network (NFV-SDN)*. IEEE. 2015, pp. 142–148.

[94] Richard Cziva and Dimitrios P Pezaros. "Container network functions: bringing NFV to the network edge". In: *IEEE Communications Magazine* 55.6 (2017), pp. 24–31.

[95] *GitHub - CPqD/ofsoftswitch13: OpenFlow 1.3 switch.* https://github.com/CPqD/ofsoftswitch13. (Accessed on 04/25/2019).

[96] Bob Lantz, Brandon Heller, and Nick McKeown. "A network in a laptop: rapid prototyping for software-defined networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM. 2010, p. 19.

[97] *Open Source - BEBA EU Project*. http://www.beba-project.eu/open-source. (Accessed on 04/26/2019).

[98] *GitHub - OpenState-SDN/ofsoftswitch13: OpenState switch implementation based on CPqD/ofsoftswitch13*. https://github.com/OpenState-SDN/ofsoftswitch13. (Accessed on 04/26/2019).

[99] Martin Nagy et al. "Integrating mobile openflow based network architecture with legacy infrastructure". In: *Information and Communication Technology-EurAsia Conference*. Springer. 2015, pp. 40–49.

[100] *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. https://iperf.fr/. (Accessed on 04/27/2019).

[101] Mohammad Alizadeh et al. "Data center tcp (dctcp)". In: *ACM SIGCOMM computer communication review* 41.4 (2011), pp. 63–74.

[102] Kuljaree Tantayakul et al. "Experimental analysis in SDN open source environment". In: *2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. IEEE. 2017, pp. 334–337.

[103] *Frequently Asked Questions - CPqD/ofsoftswitch13 Wiki - GitHub*. https://github.com/CPqD/ofsoftswitch13/wiki/. (Accessed on 05/17/2019).

[104] *OpenFlow Version RoadMap*. http://kspviswa.github.io/OpenFlow_Version_Roadmap.html. (Accessed on 04/28/2019).

[105] *NB-API-Charter-v1.1*. https://www.opennetworking.org/images/stories/downloads/working-groups/charter-nbi.pdf. (Accessed on 04/30/2019).

[106] *onos/onos.py at master - opennetworkinglab/onos - GitHub*. https://github.com/opennetworkinglab/onos/blob/master/tools/dev/mininet/onos.py. (Accessed on 05/23/2019).

[107] *sFlow-RT*. https://sflow-rt.com/. (Accessed on 06/03/2019).

[108] *Wireshark · Documentation*. https://www.wireshark.org/docs/. (Accessed on 05/06/2019).

[109] *VMware Docs Home*. https://docs.vmware.com/. (Accessed on 06/03/2019).

[110] *Running Kubernetes Locally via Minikube - Kubernetes*. https://kubernetes.io/docs/setup/minikube/. (Accessed on 06/03/2019).

[111] *Minishift - Containerized OKD Cluster*. https://www.okd.io/minishift/. (Accessed on 06/03/2019).

[112] *Testbeds – SoftFIRE*. https://www.softfire.eu/testbed/. (Accessed on 05/13/2019).

[113] *SDN Testbed | AARNet*. https://www.aarnet.edu.au/network-and-services/advanced-network-services/sdn-testbed. (Accessed on 05/13/2019).

[114] *GitHub - softfire-eu/docker-softfire*. https://github.com/softfire-eu/docker-softfire. (Accessed on 05/13/2019).

[115] *Docker Container as Mininet Host | Tech and Trains*. https://techandtrains.com/2014/08/21/docker-container-as-mininet-host/. (Accessed on 05/14/2019).

[116] *Guide On Using Docker Containers As Mininet Hosts | Talentica Blog*. https://www.talentica.com/blogs/using-docker-containers-as-mininet-hosts/. (Accessed on 05/14/2019).

[117] *MoSCoW Prioritisation | Agile Business Consortium*. https://www.agilebusiness.org/content/moscow-prioritisation. (Accessed on 05/05/2019).

[118] *GitHub - docker/docker-py: A Python library for the Docker Engine API*. https://github.com/docker/docker-py. (Accessed on 05/15/2019).

[119] *Mininet Python API Reference Manual: Main Page*. http://mininet.org/api/index.html. (Accessed on 05/15/2019).

[120] *cpython/pipes.py at 2.7 · python/cpython · GitHub*. https://github.com/python/cpython/blob/2.7/Lib/pipes.py. (Accessed on 05/15/2019).

[121] *Extreme Programming: A Gentle Introduction.* http://www.extremeprogramming.org/. (Accessed on 06/03/2019).

[122] *cpython/subprocess.py at 3.7 - python/cpython - GitHub*. https://github.com/python/cpython/blob/3.7/Lib/subprocess.py. (Accessed on 05/19/2019).

[123] *Mininet and onos.py workflow - ONOS - Wiki*. https://wiki.onosproject.org/display/ONOS/Mininet+and+onos.py+workflow. (Accessed on 05/18/2019).

[124] *Visualizing software defined network topologies using POX and Gephi | Open-Source Routing and Network Simulation*. https://www.brianlinkletter.com/visualizing-software-defined-network-topologies-using-pox-and-gephi/. (Accessed on 05/24/2019).

[125] *vis.js - A dynamic, browser based visualization library.* https://visjs.org/. (Accessed on 06/05/2019).

[126] *SDN Narmox Spear*. http://demo.spear.narmox.com/app/?apiurl=demo!/mininet. (Accessed on 05/24/2019).