Service Provisioning in SDN using a Legacy Network Management System

DAVID M. VAN 'T HOF
Service Provisioning in SDN using a Legacy Network Management System

David M. van ’t Hof

September 20, 2016
Abstract

Software Defined Networking (SDN) has become increasingly popular in combination with Network Function Virtualization (NFV). SDN is a way to make a network more programmable and dynamic. However, in order to create a homogeneous network using this concept, legacy equipment will have to be substituted by SDN equipment, which is costly. To close the gap between the legacy world and SDN, we introduce the concept of a legacy Network Management System (NMS) that is connected to an SDN controller to perform service provisioning. This way, the NMS is capable of configuring both legacy as well as SDN networks to provide customers with the services that they have ordered, while still allowing for new SDN features in the SDN domain of the network.

The main service we wish to provide using SDN is Service Function Chaining (SFC). Service provisioning consists of dynamically constructing a path through the ordered network services, in this case Virtual Network Functions (VNFs). This thesis focuses on the SDN controller and its interaction with the NMS. This project aims at configuring OpenFlow rules in the network using an SDN controller to perform SFC. Moreover, the focus will be on how to represent an SDN element and a service function chain in the legacy network NMS. The thesis also contains a discussion on what information should be exchanged between the management software and the controller. The management software used is called BECS, a system developed by Packetfront Software.

Integrating SDN in BECS is done by creating a proof of concept, containing a full environment from the low level network elements to the NMS. By using a bottom-up approach for creating this proof of concept, the information that BECS is required to send to the SDN controller can be identified before designing and implementing the connection between these two entities. When sending the information, the NMS should be able to receive acknowledgement of successful information exchange or an error.

However, when the proof of concept was created a problem arose on how to test and troubleshoot it. For this reason, a web Graphical User Interface (GUI) was created. This GUI shows the number of packets that have gone through a VNF. Because it is possible to see how many packets go through a VNF, one can see where a network issue occurs. The subsequent analysis investigates the impact of making such a GUI available for a network administrator and finds that the part of the network where the configuration error occurs can be narrowed down significantly.

Keywords: Software Defined Networking, Network Function Virtualization, Service Chaining, Service Function Chaining, Controller, REST, Network Management System, Virtual Network Function
Sammanfattning


Uppgiften löses genom att skapa ett proof of concept, som innehåller ett komplett system med alla komponenter från nätverkselement till NMS:et. Genom att använda en bottom-up-strategi för detta proof of concept kan informationen som BECS måste skicka till SDN styrenheten identifieras, innan design och implementation av förbindelsen mellan enheterna kan utföras. När informationen är skickad ska NMS:et kunna hämta information om huruvida styrenheten fick informationen utan fel.


I would like to lovingly dedicate this thesis to my late granddad, Martin de Keijzer. He inspired me to travel abroad and to perform well in my studies. I am certain that he would have been proud of me achieving to finish this thesis.

— Dave van ’t Hof
Acknowledgements

I would like to embrace the possibility of thanking a number of people helping me with this project. It is only logical to express my gratitude to my supervisor at KTH, Peter Sjödin, who helped me by providing a lot of useful feedback for the academic part of the project. Moreover, there are also some more people and organizations who made it possible for me to do this master thesis and helped me along the way.

Firstly, I would like to thank PacketFront Software for starting this project and providing all necessary equipment at the office. My gratitude goes especially to Henrik Johansson and Tomas Skäre for helping my colleague and me as well as possible and providing us with a lot of useful feedback.

I would also like to express my thankfulness to Acreo, and especially Pontus Sköldström and Anders Gavler for supervising the academic part of the project and providing me with information regarding important topics to write in the background.

My gratefulness also goes also to my colleague, Vasileios Chatzis Vovas, with whom I worked on this project. It was great working together with you on this awesome project.

Lastly, I would like to express my gratitude to my family, who has always supported me. Without them I am not sure I would have started the master in the first place, and whenever there was a bump in the road, I could always count on them.
# Contents

Abstract

Sammanfattning

Dedication

Acknowledgements

Table of Contents

List of Figures

List of Tables

Acronyms

1 Introduction

1.1 Background ........................................... 1
1.2 Purpose ........................................... 3
  1.2.1 Research Purpose ................................... 3
  1.2.2 Ethics and Sustainability Issues ......................... 4
1.3 Problem Definition ................................. 4
1.4 Research Plan ..................................... 5
1.5 Delimitations .................................... 6
1.6 Structure of the Thesis .............................. 7

2 Background ........................................ 8

2.1 Software Defined Networking (SDN) .................... 8
  2.1.1 SDN Background ................................... 8
  2.1.2 SDN Components .................................. 9
  2.1.3 Advantages and Disadvantages of SDN Networks over Legacy Networks .... 9
2.2 OpenFlow .......................................... 10
2.3 SDN Controllers ................................... 11
  2.3.1 General Information ............................... 11
  2.3.2 Developments in the Controller of SDN Networks .............. 12
  2.3.3 NOX/POX ..................................... 13
  2.3.4 Floodlight .................................... 13
  2.3.5 OpenDaylight .................................. 14
  2.3.6 ONOS ..................................... 16
  2.3.7 DISCO ..................................... 17
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4 Service Function Chaining</td>
<td>18</td>
</tr>
<tr>
<td>2.5 Network Function Virtualization</td>
<td>18</td>
</tr>
<tr>
<td>2.5.1 ETSI NFV Framework</td>
<td>19</td>
</tr>
<tr>
<td>2.6 Current Issues in SDN Networks related to Service Provisioning</td>
<td>20</td>
</tr>
<tr>
<td>2.7 Interapplication Communication (IAC) and Data Storage</td>
<td>21</td>
</tr>
<tr>
<td>2.8 The BECS Network Management System</td>
<td>22</td>
</tr>
<tr>
<td>2.9 Summary</td>
<td>23</td>
</tr>
<tr>
<td>3 Design Discussion</td>
<td>24</td>
</tr>
<tr>
<td>3.1 Requirements</td>
<td>24</td>
</tr>
<tr>
<td>3.2 Design Choices</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1 Virtual Network</td>
<td>25</td>
</tr>
<tr>
<td>3.2.2 Representation of an SDN Network in BECS</td>
<td>26</td>
</tr>
<tr>
<td>3.2.3 Communication between BECS and OpenDaylight</td>
<td>27</td>
</tr>
<tr>
<td>3.2.4 Visualizing the SFC for the BECS user</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Design of the Desired Model</td>
<td>30</td>
</tr>
<tr>
<td>3.3.1 Design Overview</td>
<td>30</td>
</tr>
<tr>
<td>3.3.2 Detailed Desired Model</td>
<td>31</td>
</tr>
<tr>
<td>3.3.3 Summary</td>
<td>33</td>
</tr>
<tr>
<td>4 Integrating SDN in Third-Party Network Management Software</td>
<td>35</td>
</tr>
<tr>
<td>4.1 The Chosen Topology</td>
<td>35</td>
</tr>
<tr>
<td>4.2 Implementation</td>
<td>37</td>
</tr>
<tr>
<td>4.2.1 Creating the Network</td>
<td>37</td>
</tr>
<tr>
<td>4.2.2 Implementing the SDN Controller’s Southbound Interface</td>
<td>38</td>
</tr>
<tr>
<td>4.2.3 Programming the SDN Controller’s Northbound Interface</td>
<td>40</td>
</tr>
<tr>
<td>4.2.4 Southbound Service API</td>
<td>40</td>
</tr>
<tr>
<td>4.2.5 Programming BECS’ Element Manager</td>
<td>42</td>
</tr>
<tr>
<td>4.2.6 Displaying Results</td>
<td>44</td>
</tr>
<tr>
<td>4.3 Implementation Discussion</td>
<td>45</td>
</tr>
<tr>
<td>4.3.1 Controller</td>
<td>45</td>
</tr>
<tr>
<td>4.3.2 Change Test Bouncer to a Real VNF</td>
<td>46</td>
</tr>
<tr>
<td>4.3.3 Allow Moving Hosts and Multiple Hosts on One Interface</td>
<td>47</td>
</tr>
<tr>
<td>4.3.4 Implementation Discussion Summary</td>
<td>47</td>
</tr>
<tr>
<td>5 Analysis</td>
<td>49</td>
</tr>
<tr>
<td>5.1 Analysis Goals</td>
<td>49</td>
</tr>
<tr>
<td>5.2 Data Analyses on the Network</td>
<td>50</td>
</tr>
<tr>
<td>5.2.1 Data Analysis on the Network Considering One Client Switch</td>
<td>50</td>
</tr>
<tr>
<td>5.2.2 Data Analysis on the Network Considering Multiple Client Switches</td>
<td>52</td>
</tr>
<tr>
<td>5.2.3 Network Troubleshooting</td>
<td>54</td>
</tr>
<tr>
<td>5.3 Conclusions</td>
<td>55</td>
</tr>
<tr>
<td>6 Conclusions and Future Work</td>
<td>57</td>
</tr>
<tr>
<td>6.1 Summary and Conclusions</td>
<td>57</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>59</td>
</tr>
<tr>
<td>6.2.1 Introduce VLAN Tags and Allow Network Layer Forwarding</td>
<td>59</td>
</tr>
<tr>
<td>6.2.2 Locating VNFs</td>
<td>60</td>
</tr>
<tr>
<td>6.2.3 Automatically create OpenFlow switches in BECS</td>
<td>60</td>
</tr>
</tbody>
</table>
6.2.4 Summary of Future Work .............................................. 60
List of Figures

2.1 Graphical Representation of an OpenFlow1.0 rule . . . . . . . . . . . . . . . . . . . . . 11
2.2 Developments in SDN Controllers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3 Graphical Representation of the Floodlight Controller Architecture . . . . . . . . . 14
2.4 Graphical Representation of the OpenDaylight Architecture . . . . . . . . . . . . . . . 15
2.5 Onix/ONOS Based Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.6 DISCO Concept Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.7 Bi-directional SFC (left) vs. Uni-directional SFC (right) . . . . . . . . . . . . . . . . . 18
2.8 ETSI NFV Framework Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.9 Difference between Proactive (left) and Reactive (right) triggered Rules . . . . . . . 21

3.1 Design of the BECS tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.2 SFC wizard design with checkboxes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.3 SFC wizard design with a dropbox and a table . . . . . . . . . . . . . . . . . . . . . . . 29
3.4 SFC wizard design with two tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.5 Experimental Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.6 BECS Internal Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.7 OpenDaylight Internal Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.8 Display Scenario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.9 Detailed overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.1 First Concept for Network Topology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.2 Improved Concepts for Network Topology . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.3 Optimal Solution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.4 Overview of Data Link Layer and Network Layer Addresses . . . . . . . . . . . . . . 38
4.5 Service mapping wizard . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.6 Interface with a SFC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.7 Systems’ Flow Chart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.8 Final Environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.9 Network Using a NAT or Proxy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

5.1 Test Network Set Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2 Resulting Packets through Firewall and DPI with One CPE . . . . . . . . . . . . . . . 50
5.3 Resulting Packets through Firewall and NAT using Two Different CPEs . . . . . . . . 51
5.4 Resulting Packets through Firewall and NAT having Multi-Tenancy . . . . . . . . . . 52
5.5 Resulting Packets considering Two CPEs on Different Client Switches . . . . . . . . 53
5.6 Resulting Packets considering Multi-Tenancy for Two CPEs on Different Client Switches 53
5.7 Error in BECS when the user input is wrong . . . . . . . . . . . . . . . . . . . . . . . 54
5.8 Debug NAT service down . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
List of Figures

6.1 SDN element representation in BECS ........................................ 58
List of Tables

4.1 Flows Currently Set by the Controller ................................. 46
Abreviations

AJAX  Asynchronous JavaScript and XML
API  Application Programming Interface
ARP  Address Resolution Protocol
BDDP  Broadcast Domain Discovery Protocol
BECs  Broadband Ethernet Control System
CAPEX  Capital Expenditures
CJM  Central Job Manager
CLI  Command Line Interface
CPE  Customer Premises Equipment
CPU  Central Processing Unit
CRE  Configuration Rendering Engine
DHCP  Dynamic Host Configuration Protocol
DoS  Denial of Service
DPI  Deep Packet Inspection
EM  Element Manager
EMF  Element Manager Framework
EMI  Element Manager Instance
GUI  Graphical User Interface
HTML  Hypertext Mark-up Language
HTTP  Hypertext Transfer Protocol
IAC  Inter-Application Communication
ICMP  Internet Control Message Protocol
IDE  Integrated Development Environment
IP  Internet Protocol
ISP  Internet Service Provider
JSON  JavaScript Object Notation
LLDP  Link Layer Discovery Protocol
MAC  Medium Access Control
MPLS  Multiprotocol Label Switching
NAT  Network Address Translation
NFV  Network Function Virtualization
NFVO  Network Function Virtualization Orchestrator
NIB  Network Information Base
NMS  Network Management System
OPEX  Operating Expenditures
PHP  PHP: Hypertext Preprocessor
QoS  Quality of Service
REST  Representational State Transfer
SAL  Service Abstraction Layer
SDN  Software Defined Networking
SFC  Service Function Chaining
SNMP  Simple Network Management Protocol
SOAP  Simple Object Access Protocol
SQL  Structured Query Language
TCL  Tool Command Language
TCP  Transport Control Protocol
URL  Uniform Resource Locator
VIM  Virtualized Infrastructure Manager
VLAN  Virtual Local Area Network
Abreviations

VM  Virtual Machine
VNF  Virtual Network Function
VoIP  Voice over IP
VPN  Virtual Private Network
1. Introduction

In recent years, the concept of Software Defined Networking (SDN)\(^1\) has become increasingly popular. SDN is an approach where the forwarding plane of a router or a switch is separated from the control plane and where the control plane is made programmable. By doing this, the control over the forwarding elements in a network can be centralized and since the controller is programmable, the forwarding and processing of traffic are programmable too. Thus, it adds a new level of abstraction that keeps the network manageable and evolvable.

The SDN approach is relatively new and there are still many matters that require attention and research. How does the controller communicate with the switches and routers? For this, the OpenFlow\(^2\) protocol exists, but can OpenFlow do everything that is required of it in the scenario for this thesis? How are nodes discovered? When the nodes have been found, how do we classify private hosts, network services, and forwarding elements? In the area of network services alone, there already exist many questions. For example: how is load-balancing to be done? Since multiple users can have the same service, how does the system know when a service is being overloaded, and when it overloads, how should the system react to that? These questions are only a few of the many questions that exist and for many of these issues, a clear answer has not yet been found.

In this chapter, the reader will be familiarized with the project that is carried out next to this thesis. The problem will be explained and a scope of the research will be defined. First, the background will be discussed after which a detailed problem definition will be presented. The chapter will continue with describing the purpose and the goals of this thesis. Next to that, the research plan and delimitations of the project that is carried out will be presented and discussed. Lastly, the structure of the other chapters will be provided to the reader.

1.1 Background

As said before, SDN is a concept where the forwarding plane and control plane\(^3\) are separated. By doing this, the control plane can be centralized and it can be made programmable. If this is done one new entity is created, called the controller. The controller can be centralized or distributed. In the latter case the network could be divided into domains which each have their controller and thus, the control plane becomes locally centralized. The controller can set forwarding rules or flows in the network elements using a protocol. Such protocol could be OpenFlow for example. What does the concept of a controller like this provide that is not there yet?

First of all, having a concept like this means that all the forwarding rules can be set from one central entity, which results in only one entity that has to be managed. From a management point of view, this is a lot easier than managing every router or switch in the network separately. Since
the controller can react to changes in the network automatically, it removes a lot of complexity and manual work from the network.

Secondly, since the controller is the centralized "brain" of the network, it has full information about the network nodes under its control and their topology. That information allows the controller to alter rules and configuration in switches and routers at one end of the network, based on something that happened at the other end. It can nearly instantly react to events in a network, such as a link or a node failure, and reconfigure it without any human interaction with the forwarding elements or controller.

Legacy networks can route traffic based on, for example, the number of hops or link bandwidth. However, the packets are always routed based on the same property that is specified in the protocol. Using an SDN controller, traffic can be routed based on many different things and different kinds of traffic can be sent separately over different paths.

Thirdly, using SDN in combination with Network Function Virtualization (NFV) could reduce the Operating Expenditures (OPEX) and Capital Expenditures (CAPEX) of operating networks in future. NFV is a concept where expensive boxes that perform network functions, also known as middleboxes, are virtualized into a Virtual Network Function (VNF). These middleboxes are for example firewalls, Network Address Translations (NATs) or proxies. Virtualization of these middleboxes means that there will be one or more Virtual Machines (VMs) that execute these network functions as a piece of software. These VMs can be run on ordinary servers.

By doing this, and since the network can be changed on the fly using SDN, when two of the same VNFs are barely used one can be shut down, and the network traffic can automatically be rerouted to the other. When more processing power is demanded from the VNF, a new one can be started, and traffic can instantly be rerouted. Thus, SDN in combination with NFV, one for controlling the traffic forwarding, and the other for controlling the traffic processing, brings much flexibility. Moreover, this combination could help minimizing OPEX significantly since the management of the network is automated and therefore the amount of employees required to operate the network could potentially be reduced. Besides that, the CAPEX is decreased since expensive middleboxes can now be run on a standard VM and can be shut down and fired up dynamically as well as migrated to follow dynamic changes, like e.g. user mobility.

Lastly, the combination of SDN and NFV makes it possible to perform Service Function Chaining (SFC) dynamically. SFC is creating a chain of services in the right order for the customer and executing the services in this chain. VNFs are virtual network functions, which can be ordered by customers. However, a customer may order multiple network functions and even ones that are specific or need to be called in a particular order for the customers’ applications. Also, the chain of network functions can be different for inbound and outbound traffic, called unidirectional SFC. In legacy networks, data centers contain racks of servers running network functions (all hardwired), which makes it hard to change the order of the network functions in a chain. By using SDN and NFV, this can be done dynamically and therefore the increasing demand on network functions can be satisfied.

However, there are still many issues regarding SDN that are to be solved. This project is about creating a system that can do service provisioning, using a Network Management System (NMS) and an SDN controller. It would include NFV-based SFC and distinguishing traffic flows from different hosts through one VNF. Also, the service provisioning will be done dynamically, and the order of the VNFs can be chosen as well. The research done on the rules required in the network is provided in [11].
Chapter 1. Introduction

The NMS that will be used is called Broadband Ethernet Control System (BECS)[12] and the controller that will be used is OpenDaylight[13]. These are requirements for the project. In legacy networks, BECS can communicate through different protocols with the elements. An example of these protocols could be the Simple Network Management Protocol (SNMP)[14]. The Element Manager (EM), a component of BECS, is one of the key components in the communication between BECS and the managed elements, because it will control the translation between the BECS core commands and the commands the element understands.

This section showed some general background information and advantages of using SDN. Moreover, it introduced the BECS system at a very high level to explain the next sections better, which will present the purpose of the work and the problems that have been identified during the research.

1.2 Purpose

This section will describe the purpose of the research performed in this thesis. First, the project’s purpose will be described and, after that, some ethical and sustainability issues will be discussed.

1.2.1 Research Purpose

The purpose of the research in this thesis is to find a way to integrate SDN into the BECS NMS and make BECS capable of performing SFC through SDN. As explained before, the NMS used is developed by Packetfront Software, since Packetfront Software has requested the research. Moreover, a research institute called Acreo helped with the academic part of the research.

There are several reasons why integrating SDN into BECS is preferred over just using an SDN controller. The first reason is that BECS already knows the concept of a service, whereas an SDN controller does not have such concept integrated. Therefore, if one would just use a controller, a separate module to implement services would have to be written. Since BECS already has Customer Premises Equipment (CPE) services like NAT and Voice over IP (VoIP), it is much easier to create a chain of these services within BECS. Currently, a service corresponds with setting certain parameters on a device, but this can be adapted to giving commands to an SDN controller. So because BECS already knows the concept of a service and already has different kinds of services available, it is more logical to make BECS capable of performing SFC rather than using an SDN controller only.

The second reason why it would be of general interest to integrate SDN into BECS is that BECS can already communicate with legacy equipment using the regular control plane of such devices. By integrating SDN into BECS, it would also be capable of communicating with SDN devices. This process creates an NMS which can manage legacy networks, SDN networks, and hybrid networks. Currently, the change from legacy networks to SDN or hybrid networks goes rather slow. There are multiple reasons for this, but an NMS that can manage both kinds and hybrid networks would influence decisions about using SDN equipment because it will be much easier to manage such networks in a single system.

To summarize, two goals will have to be achieved during the project.

• Integrate SDN into BECS by making BECS capable of communicating with an SDN controller
• Make BECS capable of rendering a configuration so that it can perform service function chaining.
1.2.2 Ethics and Sustainability Issues

An ethical problem that may arise is the fact that the controller controls all traffic. If it gets compromised, all communication can be eavesdropped and controlled by the adversary. Also, the adversary will be able to collect information about the network topology [15, 16]. In this case, the adversary could reroute traffic so that customers pay for services they do not receive, or they receive extra services. When an adversary gets to know all the traffic, the privacy of the users is at stake. For the proof of concept this is no issue, but when scaling up SDN to real-life networks, this certainly becomes one and needs to be taken into account. While this issue exists in legacy networks as well, an SDN controller or a BECS server has more control over the network than regular switches and routers and is, therefore, a more valuable target for adversaries.

SDN, in combination with NFV, has the potential to reduce energy consumption in the future and therefore contribute to the well-being of nature. For example, the controller can communicate with services like OpenStack that keep track of the VNFs in a network and how much these VNFs are used. When two VNFs with the same functionality are both barely used, one could temporarily be shut down. The controller can automatically adapt the routes that flows take and therefore the two VNFs can be combined.

NFV also allows the network operator to replace hardware boxes with software images. Hardware costs energy for producing the boxes and also for running them. Since VNFs are software-based, they are easier to maintain, and producing, running and maintaining them costs less energy.

1.3 Problem Definition

This section shows what problems are identified that have to be solved to make BECS capable of communicating with an SDN controller to perform SFC. Some general problems regarding SDN will be highlighted in chapter 2.

This thesis addresses the problem of how to represent SDN networks in BECS. The BECS system retrieves information about network elements and maintains the configuration of them on a per element basis. In a legacy network, all the elements have a control plane. Therefore, BECS can communicate with each of the elements separately and therefore manage them separately. However, in SDN networks, the control planes of the elements are combined into the SDN controller. Since this is the case, there will only be one entity that BECS can communicate with and that entity controls a whole network. It is of importance that the EM used for SDN is similar to EMs for legacy networks, because the BECS core counts on that. For this reason, there are limitations on how to represent SDN networks in BECS. Since BECS is designed to manage elements on a per element basis only, this brings some problems and trade-offs. Thus, one of the problems that is raised is to represent SDN networks in BECS and how to configure the elements separately.

The second problem that is addressed is that BECS does not have the ability to create a service chain. BECS has the concept of a service, but no ability to place them after each other, which is required for this project. Since provisioning services and rendering the configuration of those services is handled by the BECS core (which should not be altered for this project), a way will have to be found to create a chain of services in any wanted order using just the EM.

Next to that, another problem that exists and is subject to the research in this thesis is that there is no way of communication between OpenDaylight and BECS at this moment. Currently, there is no
protocol between BECS and OpenDaylight, so a protocol will have to be chosen or designed in order to have BECS communicate with the SDN controller. Also, there is no Application Programming Interface (API) that can be used to invoke the right processes. Therefore, an API will have to be written or extended that can be used to perform all the functionalities that are required from BECS. Moreover, SDN allows traffic to be distinguished on different header fields, rather than just IP addresses. The management software should be able to control the forwarding of traffic by communicating with the controller. In the current situation, the management software is not capable of doing so. To make the management software capable of controlling an SDN network it needs to send commands to the controller which in turn must interpret them and generate rules corresponding to these commands that can be pushed to the network. The fact that SDN networks can forward packets based on many different header fields, makes it harder to create a general API and there may be a trade-off between what kind of traffic distinguishing to support and the generality of the API. Next to that, a decision will have to be made about whether a push or pull protocol will be used.

The challenges described above will all have to be dealt with. These difficulties apply to designing and implementing the system. However, evaluating the system brings problems with it as well, which are at least equally challenging. For this project no physical elements will be available, so some way of virtualizing an SDN network has to be found. Consequently, the next problem that is addressed is to find a way to virtualize a network to evaluate the system.

When BECS has set rules in the controller, the traffic should go through the right VNFs. However, that configuration is set correctly cannot be taken for granted. So, how can this be checked and how can errors in the network be found. Some programs can read the traffic on specific interface(s) of an element in a virtual network, but doing so requires logging in to different entities in the network and run such a program. Using a program like this is cumbersome since it could require a series of actions that have to be taken by the user. Therefore, it would be better to find a way that makes pinpointing configuration errors in the network easier.

To summarize this section, a list of the main problems and challenges can be found.

- How to represent SDN networks in BECS
- How can a service chain be created in BECS
- How should communication be set up between BECS and OpenDaylight
- How can the solutions found be tested without any physical elements
- How can errors in the network be found.

1.4 Research Plan

This section will try to explain briefly the approach that will be taken on tackling the problems described in section 1.3. Since there are different sub-problems, the way they have been handled will be described per problem.

Since the required information to solve the problems of communication between BECS and OpenDaylight and representing an SDN network in BECS is not available from the beginning, this information has to be found. At this point, it is unknown what rules have to be applied to the network because there is no network yet. Consequently, a network has to be created first to find the rules.
The OpenFlow rules that have to be pushed to the network are decided upon in [11], but these will have to be programmed in OpenDaylight. OpenDaylight cannot automatically push those rules to the network. This functionality has to be programmed. Since BECS will decide on a service chain per CPE, somehow the controller has to know where the CPEs, VNFs, and the Internet are located and what route the packets should take. For this reason, a classification of hosts will have to be made.

From this point forward, issues regarding this thesis can be solved. The network and rules are known and therefore how to send information and what information to send from BECS to OpenDaylight can be decided upon. After that, the communication can be set up and the necessary API functions can be implemented.

Henceforth, actions can be taken to send variable values from BECS to the controller. To send variables to the controller, a way has to be found to create the service chain, represent SDN elements in BECS and send the right traffic. This needs to be done in an as similar way as possible to existing EMs. To represent SDN networks in BECS, there are different design possibilities, which differ in how much of the network is shown in BECS. When a solution for representing SDN networks in BECS is implemented, a way for BECS to create a chain of the services will have to be designed. Here there are different design possibilities as well. The various design possibilities are all in the EM and differ in where in BECS they are implemented, whether existing functions have to change and whether it is possible to append and order functions.

1.5 Delimitations

To perform research correctly, a scope has to be defined and border lines have to be drawn to decide what falls within the scope and what does not. This section explains what these border lines are.

The research carried out will be on BECS, the controller, and the communication between these two entities. However, BECS is an extensive program with many different modules within its core. Altering this core could have a significant impact and therefore it is a requirement that the core will not be changed. For this reason, only an EM and possibly other interfaces will be altered in this thesis and not the BECS core. The controller falls within the scope of the thesis as both a northbound interface as well as a southbound interface will have to be used to send the right commands to the network. However, OpenDaylight consists of many modules, each providing their functionality. It is impossible to get to know each and every module in the given period of time as there are new modules created every day. For this reason, only the core modules will be used.

What will not be discussed in this thesis is the security that might be required. The controller could be compromised for example, or even more closely related to the topic, the traffic could be eavesdropped. Security on SDN controllers is a complex issue and a whole different research could be conducted on this subject. Next to that, Packetfront Software requested that this topic was not researched. For this reason, it is out of scope.

Next to that, the problem of host classification will not be discussed in much detail. However, some information will be required to understand some of the choices made. In a network, there may be multiple types of hosts. Host classification is the process of determining what the type of each host is. This topic will be explained in more detail in [11].
Anything regarding traffic forwarding will not be within the scope of this thesis. It is evident that traffic forwarding has to be taken into account in the project and that knowledge about this topic is required. However, this is described in great detail in [11]. At some points, some high-level information will be provided to justify choices made.

VNFs are complex pieces of software. For this reason, only some high-level background knowledge is provided in the next chapter. Throughout the thesis, it is assumed that VNFs are there and are set up. They will be treated as black boxes which can receive and send back packets, because the thesis is not about how VNFs are set up, what processes are running inside a VNF or how to manage them.

1.6 Structure of the Thesis

In this section, the outline of the rest of the thesis will be explained. The next chapter will contain background information about the specific elements in SDN, including the controller and OpenFlow. Next to that, research is conducted regarding what information should be stored where. Information regarding developments in SDN controllers and about protocols is provided. It will also contain information about BECS and inter-application communication (IAC), since those topics are important to fulfill the project. Even though VNFs are not a part of this thesis, it is necessary to discuss some background information on them, because the controller needs to be able to find where VNFs are located. Therefore chapter 2 will also contain information about this subject. Another subject discussed in the same chapter is the concept of SFC.

Chapter 3 will explain the requirements for the project in more detail. Moreover, when the requirements are explained, different parts of the design are discussed and design choices made are argued for. Chapter 3 ends with a brief overview of the design.

Chapter 4 will show what has been done to draw conclusions and make choices. How specific parts are implemented will also be explained in this chapter as well as what effect the requirements from chapter 3 have on the design. Difficulties found during the creation of the proof of concept will be provided to the reader and solutions of how was dealt with these difficulties will be explained.

Chapter 5 will critically analyze the data that is collected and show the results. As will be seen later the results consist of basically two different parts. The first part will show the effects of the commands sent from BECS through OpenDaylight to the network. The second part analyses how well the web GUI created for the project helps the user troubleshooting the network. After that, the results will be discussed.

Lastly, in chapter 6 a summary will be given and the results will be shown. What conclusion is drawn will depend on the data collected and after the conclusion the future work will be discussed.
2. Background

In this chapter, background information about the project done will be discussed. First, SDN will be introduced in more detail in section 2.1. Then, in the subsequent paragraphs, the components that are needed to set up an SDN network will be presented to the reader, starting with the OpenFlow protocol in section 2.2. The subject of section 2.3 is SDN controllers and the different developments that have been done on those. These developments and the different controllers will be explained in more detail. Section 2.4 will discuss SFC in more detail. After that, NFV will be introduced in section 2.5.

When the reader has been familiarized with the different components of SDN, some issues regarding SDN will be discussed in section 2.6. After that section 2.7 will be about the topic inter-application communication and data storage, since that will be an important topic later on in the thesis. The BECS system will then be explained to some extent to the reader in section 2.8 to be able to understand the rest of the thesis. The chapter will conclude with a summary in section 2.9.

2.1 Software Defined Networking (SDN)

In this section, SDN will be explained in more detail. First, a small historical background will be provided. Thereafter, the components needed to deploy an SDN network will be discussed. Each of them will be considered in more detail next sections. Then, the advantages and disadvantages of SDN networks over legacy networks will be discussed.

2.1.1 SDN Background

Until a few years ago, all routers had a control plane and a forwarding plane. The control plane makes decisions based on the routing protocol used. The forwarding plane checks the routing table when an incoming packet has been received and finds the correct next hop, so it can send the packet there. This means that all computing is done inside each of these nodes in the network. However, in large networks, some problems occur if a route needs to be changed manually. A person who knows the network has to log in into every router that affects the route and alter the routing table or configuration of those routers, typically using the Command Line Interface (CLI) commands[17]. Manual changes leave mistakes, because humans are not infallible, resulting in traffic being processed incorrectly by the network.

In SDN the control planes are taken away from the nodes and are locally centralized into one SDN controller, which is also made programmable. The controller has an interface that can be used to send commands to it (the northbound interface) and it has an interface that is connected to the network (the southbound interface). By centralizing the control plane this way, the network is made more abstract to the northbound interface of the controller. When a person alters something in the network by issuing a specific command on the northbound interface of the controller, it could automatically change the rules in the forwarding tables of the associated switches. However, since the controller is programmed, it will only do the routing in the way it is programmed to. Since the network is abstract,
the person altering the network does not necessarily need to know as much about configuring routers or switches as before. Next to that, the fact that the forwarding rules in the switches can be changed by programming the controller adds a lot of simplicity and flexibility, but it requires programming skills.

The result of this is that routes all over the network can be changed nearly instantaneously and, maybe even more important, more accurately (with less erroneous routes). Since it is a program changing the flow tables in the OpenFlow switches, there will be significantly fewer mistakes in the network configuration. If there is a bug in the program, it can be solved more easily since the location of the bug can be found by using standard debugging techniques.

Also, SDN allows network resources to be used much more efficiently. The controller can be programmed to set flows automatically, but also to remove redundant or old flows. A side note to this is that one has to develop an algorithm that could find old or redundant rules to make it work. Another option is to use the time-out functionality when setting a flow. This way flows will automatically be deleted after a specific period of time.

2.1.2 SDN Components

An SDN network requires different components. First of all, switches are needed that can interpret commands from a centralized controller. In the current state of technology, these switches are not regular layer 2 switches. They can do much more than a regular layer 2 switch. For example, they can inspect a packet on network layer and transport layer headers. Furthermore, they can alter e.g. the Internet Protocol (IP) header and transport layer ports.

A controller is required next to the switches to enable the altering of the forwarding rules inside each switch. It needs to be capable of noticing that nodes have joined or have left the network. Besides that, it should be able to understand the topology, because if one node fails it has to set new paths avoiding that specific node. In order to get all this information from the controller to the switches and routers, communication is needed and thus a communication protocol.

Without protocol no communication is possible. The most common protocol in SDN networks for communication between switches and the controller is OpenFlow. OpenFlow is an open standard protocol which defines a match and an action. If a packet matches an entry in the match table, the corresponding action is performed.

2.1.3 Advantages and Disadvantages of SDN Networks over Legacy Networks

There are many advantages when using SDN. The most important advantage is the programmability and dynamicity of the network and the ability to add new features. Because the network becomes programmable, it is no longer necessary to change rules in each switch manually. By automating the change of rules and using a northbound interface, with possibly a Graphical User Interface (GUI), the involved routes can be altered in an easier way than before. At worst, it requires some additional programming in the controller, but even then that would take less effort than logging into all involved routers to change the configuration as needs to be done in legacy networks.
Because the SDN controller holds information about the whole topology, it has another great advantage. Normally if a router fails, only its neighbours notice this directly, and can directly change the traffic avoiding the failing node. It might take several iterations of routing protocol message exchanges before the network converges to a new state. Using an SDN controller, a change in the network can be made somewhere else rather than where the failing node is situated, and the change can be made nearly instantaneously. In some cases rerouting traffic somewhere else in the network results in an overall better solution than when the change is made locally.

Another advantage of SDN networks is that network resources can be used much more efficiently. The network can be maintained easier using a centralized controller. Maintenance of the network includes rerouting of traffic which can be done relatively quick compared to legacy networks. This property is important if the network resources have to be used efficiently. When using VNFs, as will be explained later in this chapter, and virtual switches, the property of having rerouting done almost instantaneously becomes very powerful.

Something that might cause problems in SDN networks which is not present in legacy networks, is the centralized controller. By centralizing the controller, the network becomes dependent on a single entity. If this entity breaks down, the currently installed rules in the network will still stay active since after installing them in the switches the controller is not involved any longer. However, new sorts of traffic will not be handled, because the switches do not know what to do with it. So the switches try to send requests to the controller, which is broken, and therefore they drop the packets. Having a centralized controller brings another risk. If an adversary succeeds in taking over the controller, he or she can eavesdrop all traffic or alter it. Also, there are ways to perform a Denial of Service (DoS) attack on the controller if reactive rules are used. A very simple solution could be to use proactive triggered rules only, but that would reduce the flexibility of the system since the controller will be unable to react to changes in the network. Currently, some scientists try to address these security issues in SDN[18].

2.2 OpenFlow

To understand the concept of SDN, the reader is introduced to the OpenFlow protocol. This section is dedicated to explain this concept.

OpenFlow is a standardized protocol for SDN created by the Open Networking Foundation (ONF)[19]. The OpenFlow protocol is used for communication between the controller and the different switches in the network (i.e. the southbound connection). OpenFlow does not provide any services for a possible northbound connection. The controllers researched in the next section can all communicate via OpenFlow. As pointed out earlier, an OpenFlow rule consists of a match and an action. This rule is sent to the switch according to the JavaScript Object Notation (JSON) standard. The match can be made on different and multiple properties of a packet as well as on various layers. Figure 2.1 shows on what a packet can be matched.
As can be derived from figure 2.1, a packet can be matched not just on destination IP-address, but also on different fields of the layers 2 to 4. This property makes routing in SDN a lot more flexible. Packets with for example a higher priority could be routed through a different way than packets with a low priority to provide Quality of Service (QoS). Traffic could also be forwarded based on the program that uses that traffic[21], which enables adding different network functions to packets from different applications on the same host. Besides the match and action, also statistics are kept in the flow entry. By doing this, it could even be possible to route traffic based on statistics if the controller has to.

Next to the flexible forwarding, using OpenFlow has another advantage. It can also manipulate packets. This property results in the OpenFlow switch being capable of for example adding or removing VLAN-tags to or from a packet and change its Ethernet and IP-header. New versions of OpenFlow can also change Internet Control Message Protocol (ICMP) headers and Address Resolution Protocol (ARP) headers[22].

### 2.3 SDN Controllers

To separate the forwarding plane from the control plane, an SDN controller is required. This section will provide the reader with general information about SDN controllers. After that, a selection of controllers has been made to be discussed in more detail, and comparisons are made. The main focus will be on the recent developments in SDN controllers.

#### 2.3.1 General Information

The SDN controller is, as the name already suggests, the "brain" in the SDN network. Because it controls all the routing in every node in the SDN network, it collects all the information about the topology. This is different from most legacy networks where, depending on the routing protocol, the nodes do not know the network topology. Because of this difference, if something happens in legacy networks at one end of the network, there cannot be an as quick reaction at the other end as in SDN networks, because the failure has to propagate through the entire network.
In order to communicate with the nodes, the controller needs a protocol. The most well-known protocol is OpenFlow. However, there are also other protocols, but since OpenFlow is the most popular protocol and the open standard that is most widely supported, only OpenFlow has been discussed. OpenFlow uses a Transport Control Protocol (TCP) connection with the nodes\[19\][23].

New nodes may be added to or removed from the network. The controller has to have a way to adapt to these situations. Otherwise, the routing done by each node may be incorrect. Therefore, SDN controllers have a node-discovery mechanism, which differs for every controller. To keep track of the so-called “network state”\[19\], the SDN controller often has a database in which nodes and links are stored.

**2.3.2 Developments in the Controller of SDN Networks**

Since the start of developing SDN networks, much progress has been made regarding the implementation of the SDN controller. In the first controllers, the programmer had to write raw OpenFlow protocol commands in the program. Later, a layer of abstraction was developed for the programmer, so it was no longer necessary to write these raw commands. This layer of abstraction made it easier for developers to program the network.

Recently developed SDN controllers are capable of communicating between different controller instances, and the network topology is shared between these different instances. Using the shared topology database multiple controller instances could invoke commands on one network.

![Figure 2.2: Developments in SDN Controllers](image)
In large networks, a single instance of a controller managing everything may lead to the controller being overloaded. Therefore, in future and recently developed SDN controllers a controller can manage a part of the network and can communicate with other controller instances. By duplicating and/or distributing the controller, the possibility of a controller being overloaded is no longer an issue. The duplication of controllers may also be used to add redundancy. Figure 2.2 shows the developments over time, where the left shows the oldest controllers. Examples of controllers are NOX/POX and Floodlight(A), OpenDaylight(B), ONOS(C) and DISCO-based controllers(D). These controllers will be presented in a bit more detail in the next sections.

2.3.3 NOX/POX

The NOX controller is one of the very first SDN controllers. After NOX, also POX was created. POX is the same controller architecture as NOX but written in a different language, namely Python. Only POX will be taken into account for the rest of this section since the controllers are essentially the same. POX supports OpenFlow1.0 [24], but it has been extended to be able to use different protocols according to [19]. POX has the functionalities for discovering the topology and performing host tracking. However, POX does not provide many other features. When one has to push flows to a network, the flow definitions have to be written in a JSON string[25]. Unlike controllers discussed in later sections, there are no flow objects that can be pushed down to the network.

**Messenger** - In POX the messenger is the block of code that handles the communication between POX and external processes[25]. The messenger is implemented by so-called services[25]. Different existing services can be used and new services can be written as well. An example of a service is the messenger.log service, which is the service that communicates with the log files. Communication is implemented by so-called transports. Standardized transports are available for Hypertext Transfer Protocol (HTTP) and TCP, but new transports can also be added by extending the POX controller[25].

2.3.4 Floodlight

The Floodlight project is a project started by Big Switch Networks[26]. The Floodlight SDN controller is written in Java[27]. It can be integrated into Eclipse, which is an Integrated Development Environment (IDE) and it is therefore relatively easy to add modules to it. Floodlight supports a partly non-OpenFlow network as long as there is no loop created in the topology[28]. Because of this, part of the network can be a legacy network. Moreover, Floodlight supports OpenStack[27], which might be necessary in future if NFV is done in data centers.

**Device Manager** - The Device Manager is responsible for the discovery of the devices. It distinguishes devices based on Medium Access Control (MAC) address and Virtual Local Area Network (VLAN) tags. However, besides those attributes also the connecting switch, that switch’s port as well as the IP address of the devices are learned[29].

**Link Discovery Manager** – The Link Discovery Manager learns about connections within the OpenFlow network. To do so, it uses Link Layer Discovery Protocol (LLDP) and Broadcast Domain Discovery Protocol (BDDP) packets[30]. The latter protocol is used to discover links that are partially outside the OpenFlow network.
Figure 2.3: Graphical Representation of the Floodlight Controller Architecture[31]

Figure 2.3 shows the architecture of Floodlight. Some important blocks of the architecture will be explained in more detail below.

**Topology Manager** - The Topology Manager of Floodlight uses the Link Discovery Manager to form a virtual topological graph of the network. It uses the idea of so-called OpenFlow islands[32]. Such an island is a part of the network that consists of OpenFlow devices. OpenFlow islands can be interconnected by legacy network equipment. The Link Discovery Manager learns about links between OpenFlow islands via the BDDP packets, and the Topology Manager can then create these virtual OpenFlow islands[32].

Some concerns are that the Floodlight project only supports OpenVSwitch[33] and Indigo VSwitch[34] as virtual switches[35]. Also, the current version of Floodlight dates back to October 2012[36], which is more than two years ago at the time of researching this controller. Despite the fact that parts of this controller are used in other implementations as will be shown in section 2.4.7, some doubts arise about its current activity.

### 2.3.5 OpenDaylight

OpenDaylight is an open source SDN controller, which is built in Java like Floodlight. The project was initially founded by Cisco, Ericsson, Jupiner, HP, Microsoft, VMware and many other companies[37]. Because of the amount of businesses that are participating in this project, the project is widely supported by the industry. OpenDaylight supports multiple southbound protocols to communicate with the switches, amongst which the popular protocol OpenFlow. It also has a web-based GUI to make it easier for users to add, remove or change flows in the network.
Figure 2.4: Graphical Representation of the OpenDaylight Architecture[38]

Figure 2.4 shows the setup of OpenDaylight. There are six main network service functions. Since OpenDaylight consists of many packages, only the main features will be discussed below.

**Host Tracker** – The Host Tracker service of OpenDaylight is used to keep track of what hosts are currently active in the network. The host tracker has an internal database containing the MAC and IP addresses, the switch to which a host is connected (including port), and possibly a VLAN tag. Next to this, the host tracker has possibilities to notify OpenDaylight extensions when a host has become active or inactive[39].

**Topology Manager** – The Topology Manager is the entity that keeps track of connections (called Edges in OpenDaylight) between the switches. It does not have information about the hosts or the switches themselves but only knows what port on what switch is connected to what port on another switch. For this reason, it has knowledge of the topology of the network. To discover the topology, the topology manager sends out LLDP packets[39].

**Switch Manager** – The Switch Manager keeps track of the switches, called nodes in OpenDaylight. All details are stored upon discovery[39]. After that other packages, also called bundles, can use the information about the switches by calling specific functions on a Switch Manager object.
Service Abstraction Layer (SAL) – One of the main innovations of OpenDaylight is the SAL. The SAL is a layer between the OpenDaylight main features and the network. OpenDaylight uses different protocols. The SAL is there to translate commands and requests from the controller into the necessary protocols. Besides that, it also gets the information from those different protocols and translates them into something that the corresponding OpenDaylight modules can understand. Among the protocols that can be used are different versions of the OpenFlow protocol. The actual flow programmer is part of the SAL.

2.3.6 ONOS

ONOS is the open source SDN controller that is inspired by the Onix controller. ONOS controllers can be divided among multiple servers that run different instances of the same controller. The controllers in previous sections cannot be distributed and communicate between the different instances. The fact that ONOS can do this adds redundancy to the ONOS controller, which is a special feature of ONOS. The controller instances can be placed on multiple servers and maintain a connection between each other to share the topology[40]. When a switch connects to the network, the different ONOS instances it is connected to decide on which one will have the responsibility for that switch. The ONOS instance that becomes responsible for maintaining the switch is called the master. This decision is made using Zookeeper. Figure 2.5 shows the architecture of an Onos/Onix controller.

![Figure 2.5: Onix/ONOS based architecture](image)

ZooKeeper – ZooKeeper is an Apache service that is used to synchronize applications that are distributed[42]. Since ONOS controllers can be spread out over different servers, they will need to have synchronization about which instance is the master of what switch and what the network state is. An instance has to contact the ZooKeeper to get master privileges over a switch. ZooKeeper makes sure that only one instance of the ONOS controller can be the master. The network state is saved in the Network Information Base (NIB), which is also kept synchronized by ZooKeeper.

The planned release for ONOS was at December 5th 2014[43]. At the moment the ONOS controller has indeed been released and will be updated every three months.
2.3.7 DISCO

The term DISCO stands for “DIstributed SDN COntrl plane” [44]. Please note that DISCO is not one specific controller, but rather a proposed architecture that could be built upon already existing controllers. The concept of DISCO is very experimental and only used in [44] so far. In [44] Floodlight was used to implement a DISCO controller. As said in section 2.3.2, DISCO controllers are controllers which are capable of communicating with each other on the application level. This kind of communication is called inter-domain communication. The main information exchanged in inter-domain communication is what hosts are available in the domain. However, the topologies of domains are not shared.

The inter-domain communication is the communication between the controllers. In [44] this communication involves the messenger and the agents. The messenger is responsible for maintaining a stable communication between the SDN controllers. Through the messenger, different agents can use the channel to communicate with the other known controllers. This way newly discovered hosts will be communicated to other points in the network and also connection set-up and tear-down can be done. Similarly, the link information can be sent to the other controllers inside the network.

The controllers will communicate with the domain they are controlling as well, which is called intra-domain communication [44]. The intra-domain communication involves the communication that manages the network which is controlled by the controller. This includes storing flows in network elements, discovering them and in the case of e.g. a malfunctioning link, adapting the network to these malfunctions. Figure 2.6 shows the basic architecture of the DISCO concept.

![DISCO concept architecture](image)

DISCO enables division of a network over multiple controllers. This can add redundancy in case the sub-networks overlap, but in large networks, it also reduces the amount of traffic going to one controller significantly. However, this may remove one of the advantages of SDN, which is that one
can program the whole network in one single point rather than logging into different routers to add the routing tables. The SDN controllers may require different programs and therefore a program may have to be developed for each controller.

In contrast to ONOS/Onix controllers, DISCO controllers advertise the domains they are peering with and a list of reachable hosts in their domain[44]. In ONOS/Onix controllers, the different controller instances share the same database that contains all network-wide information like topology and hosts.

### 2.4 Service Function Chaining

In chapter 1, the term SFC has been shortly discussed. This section will go a bit deeper into this subject. SFC is the creation of a path from source to destination that goes through specific network functions, like NATs, Deep Packet Inspections (DPIs), and firewalls[9]. This section provides the reader with an insight on SFC in SDN/NFV networks.

In SFC there are two kinds of service chaining, namely uni-directional and bi-directional service chaining. Uni-directional SFC is when the service chains inbound and outbound differ from each other. Bi-directional service chains are service chains where inbound and outbound traffic follows the same chain, but in reverse order. Figure 2.7 shows the difference between these two kinds of SFC.

![Figure 2.7: Bi-directional SFC (left) vs. Uni-directional SFC (right)](image)

By using SDN, service chains can not only differ per host or subnet, but also per process or application. These differentiations are possible because OpenFlow rules can be based on many more packet properties than legacy forwarding rules. Moreover, the fact that OpenFlow rules can be changed nearly instantaneously using an SDN controller, results in the possibility to set up, alter and tear down these service chains quickly.

### 2.5 Network Function Virtualization

Network Function Virtualization (NFV) removes the need for network functions or middleboxes to be in a specific hardware box. Instead of having specific boxes for e.g. NATs, firewalls, and other network functions, the concept of NFV allows these functions to be placed in one or multiple Virtual Machines (VMs). The result is a Virtual Network Function (VNF) for each kind of middlebox. By doing this, costs can be reduced for companies hosting this kind of network functionalities, because they can run anything on common off-the-shelf servers instead of buying expensive boxes. One should keep in mind...
that network operators are already facing challenges regarding energy consumption and space for the
different boxes. If they intend to launch a new service, it may be hard for them to find the space
to put the box that offers that service[45]. Moreover, hardware can degrade and become outdated.
By using images of network services, this problem could be addressed. When a server becomes too
old, it can be replaced and the image(s) that were installed on it can easily be placed on the new server.

Next to cost reduction NFV contributes to flexibility and redundancy. When a service is needed,
an instance of this service can be fired up on a host machine that is already there. If the service is no
longer needed after a while, the instance can easily be shut down again. Besides that, it is possible to
run multiple VNFs on a single host. These VMs can be copied to other host machines, which would
add redundancy for all the services offered by those VNFs. This also induces the possibility to check
whether a service has been overloaded and react to that. If a specific VNF is overloaded, a new VM
with the same service can be fired up and activated as long as necessary to handle a part of the traffic
that requires the service the VNF offers.

In combination with SDN NFV can be even more powerful. The SDN controller can, together with
the Virtualized Infrastructure Manager (VIM) and an NFV Orchestrator (NFVO), control firing up
new VNFs, reroute traffic and perform load balancing, which makes networks more flexible. In such
case the VIM and NFVO take care of firing up VNFs and the SDN controller takes care of rerouting
the traffic. It has the potential to reduce a lot of power consumption and since there will be no
instances running that are barely used or not used at all, the amount of hardware that is turned on
without being used will be reduced significantly as well.

Another new concept that comes with NFVs is the term VNF chaining, which is a specific kind
of Service Function Chaining (SFC). In legacy networks, data centers contain big racks of network
functions that are connected through a wire. However, what if the customer wants to order a new
service and place it in the middle? Alternatively, what if the customer decides one of the services has
become unnecessary? Many changes would have to be made within the rack to perform this change.
The concept of VNF chaining could address this problem. In VNF chaining the network functions
are virtual. Because of this, the changes in the racks become simple routing changes between VMs.
As seen before, one of the powerful things of SDN is that it makes it possible to adjust the network
configuration quickly with minimal errors. The result is a very powerful combination of NFV and
SDN that reduces the amount space required in data centers, increases the flexibility and scalability
of networks and could minimize the energy consumption.

2.5.1 ETSI NFV Framework

Telecom companies have been looking into NFV for a couple of years now. In 2012 they started
together with ETSI a working group, called ISG NFV[46]. The ETSI NFV framework is one of the
results of this co-operation. The framework consists of three different parts[47]. Figure 2.8 shows a
high-level view of this framework.

The first part consists of the VNFs. These are regular network functions that one can also buy
in hardware boxes, but they are implemented in software. Since it is software, they can be extended
to do more or less than legacy middleboxes. The NFV Infrastructure contains a virtualization layer
on which the VNFs are run. It virtualizes the hardware resources to create virtual resources. Then
the third and last part controls the firing up and shutting down the VNFs and manages the resource
availability during the VNFs life-cycles[47].
2.6 Current Issues in SDN Networks related to Service Provisioning

Since SDN concept is still young, there are still many unsolved issues. One of the issues is that one could create a Denial of Service (DoS) attack on the controller in some cases. There are two types of triggering rules in SDN, reactive and proactive triggering. When using proactively-triggered rules, the controller initiates pushing the rules itself, based on the way it is programmed. When talking about reactively triggered rules, this is not the case. Using these rules, the controller reacts on packets that come from the network in order to decide on the rules. Figure 2.9 shows the difference between the two types of triggering. On the left side, proactive triggering is shown, which can be initiated by another entity or the controller. On the right side, reactive triggering is shown, which is based on an event in the network. A Denial of Service (DoS) attack can be launched on the controller if it uses these reactively triggered rules. For example, if someone can fake packets that have to go through the controller. Researchers are currently trying to address this problem[18].

Another issue that arises is host classification. The controller knows the switches and the hosts in the network, since switches are found through the LLDP protocol and hosts will be found as soon as they send the first packet to the switch they are connected to[48]. Because there are no flows in the switch for that packet yet the controller will receive the packet and therefore discover the host. However, the controller cannot distinguish a customer end-host from a web server or from a VNF that is a firewall. So what if, for instance, an Internet Service Provider (ISP) would like to provide customer x with a firewall, since that is what x ordered? It will need to specify for example the Internet Protocol (IP) address of the firewall and also the IP-address of the CPE, the hardware that is present at the involved customers location. However, these addresses may change over time. The service can be moved, temporarily shut down or reset. Then hard-coding the address does not work. So there should be a look-up service for VNFs that can tell to the controller where for instance a firewall can be found.
A problem that could also arise is when someone would like to have a third party NMS to communicate with the controller to send the configuration of the network. Current software can communicate with legacy routers and switches, but is unable to communicate with the SDN network elements. This problem occurs because these NMSs communicate with the control plane of the legacy routers to change or read the configuration and these new network elements, called OpenFlow switches, do not have a control plane. A solution could be to make the NMS capable of communicating with the controller through an API. However, neither the NMSs nor the controllers have these specific APIs.

Because multiple CPEs use the same services, it may happen that one or more services get overloaded. So how does the controller notice that a specific service is being overloaded or is close to being overloaded? And how should the system react to that? Should it somehow fire up a new virtual machine with the same service? Should it redirect traffic to another VNF that has the same service on it? Or, perhaps, should the system just let the customer know a service is currently unavailable?

Lastly, but certainly not the least problem in SDN, is the issue of multi-tenancy of VNFs and SFC. If multiple CPEs are using the same service, how will traffic from one CPE be separated from the other when the traffic returns from the VNF? Moreover, what about the fact that a customer can order multiple services? How could these services be placed after each other as a service chain?

Overall, there is still a lot to do regarding the development of SDN. This section only pointed out a few of the many issues that still need to be solved.

### 2.7 Interapplication Communication (IAC) and Data Storage

The term Inter-Application Communication (IAC) is used for methods to efficiently and effectively exchange information between programs. These methods are the base for application integration and communication. According to [49], there are two main problems on this topic. The first one consists of the exchange of information and data between different applications or processes. The second one is that all the different applications have to understand the mutual process [49]. This understanding comes down to an efficient communication of information.

However, when data is exchanged, also what is the best place to store it needs to be considered. For example, a third party software that manages the network might not need to know on which port
of a specific switch a host can be found. In this case, this information does not need to be shared by the controller. In the case of SDN little research has been done on communication with third party applications. The main focus from the research institutes seems to be on optimizing the abstraction of the northbound API for the programmer and communication southbound, to the network elements.

### 2.8 The BECS Network Management System

The BECS system is a multi-vendor NMS designed by Packetfront Software. It aims to make the underlying network abstract to those who need to make changes to the system. BECS has an interface that is connected to a GUI in which changes to the network configuration can be made. Currently, another interface is connected to the control plane network elements. It can automatically add, remove and alter services that are ordered by a client. It is also connected to all end-routers in the network. For this reason, BECS knows at least a part of the network topology and in some cases the whole topology.

There can be different types of routers and switches in the network that need communication with BECS. Connecting to all those network elements, called elements in BECS, is not easy. All vendors use different syntax, commands and protocols to change the configuration, and therefore BECS needs so-called Element Managers (EMs) that translate commands into a language that a specific element understands. Because every specific brand or product type has its vendor-specific commands, BECS needs an EM for every brand and product type. BECS uses a tree structure of amongst other things elements, configuration, services, and resources. Elements have services under them, which could, for example be a Virtual Private Network (VPN) or VLAN service. The service typically applies to a port on the element. Then there are resources which can be under the service, but also under the element. An example of a resource is a static IPv4 address. When something is changed under or in the element, BECS sends the new configuration through the EM to the element. In case multiple elements have a configuration that is changed, this is done per element.

Next to the EM, BECS has a Configuration Rendering Engine (CRE). This entity of BECS interprets the changes made via the GUI or API and translates those into a syntax and commands that the EMs understand. After the EM gets the commands, it translates them into something the particular router the EM is written for can understand. The configuration is sent from the CRE to the EM for each element, in case it changed. An element is a switch or a router. Under switches and routers there are interfaces. Below the interface there can be so-called ”service attaches” for various services that should be applied to a CPE.

When a customer, for example, calls a telecom company to order a firewall service, the employee working at the customer service desk can add this service to the customer via the GUI. The BECS system then communicates with the element present at the customer, the CPE, and those in the network to enable the new firewall service for that customer.

Customer service employees usually do not have enough technical background to be allowed to alter configuration on routers. By having an NMS like BECS, the company does not have to hire employees to log into each router remotely to alter configuration in the element. Therefore companies can cut on the costs, and human errors can be avoided.
2.9 Summary

This chapter gave the reader a brief introduction to SDN. It discussed the key elements of a SDN network. After that, a historical view has been provided on the developments of SDN controllers. Some of the controllers may have some relevance later in the project, because of the way they are designed. Since this thesis is mostly about the communication on the northbound of the controller i.e. the communication with BECS, the DISCO solution might be of interest since it communicates directly between the controller instances.

Some information about SFC has been provided because this is one of the main topics of this research. It was shown that there are different kinds of service chains and that using SDN, these kinds of service chaining can be exploited even more. After that, NFV and the VNFs were discussed. NFV is a very powerful tool to increase flexibility, scalability, and redundancy of the network. It can reduce costs, human errors and energy consumption drastically and is universal to any kind of VM that could be used. The ETSI NFV framework has been discussed at a high level as well.

The section “inter-application communication and data storage” showed that two things need to be taken into account when two programs need to communicate. The first is the fact that the data exchanged should be minimized. Therefore, a decision has to be made about what information will be sent to other entities. The second task is to find out where data should be stored. Data should be stored in as little copies as possible and reducing the traffic between applications might result in storing more copies of the data.

Lastly, the BECS system was introduced to the reader. BECS does service provisioning through a GUI. It has different EMs that communicate with various brands and types of routers and switches. The CRE processes the changes made through the GUI and passes them on to the EMs to communicate them to the network elements.
3. Design Discussion

During the project, different design choices had to be made. First, this chapter will show the requirements. There were trade-offs and different options for parts of the design. This chapter goes into detail about the decisions made, but also what the consequences are of the decisions made and what the advantages would have been if other options had been implemented. Additionally, a brief design overview is provided at the end of the chapter.

3.1 Requirements

To produce a working program, first the requirements need to be defined. As for most projects, this one has some requirements as well. This section is dedicated to the requirements set for this thesis project. However, some requirements will introduce some restrictions.

To make the project valuable for PacketFront Software, the company has given the requirement that OpenDaylight must be used for the implementation. OpenDaylight is a very extensive program that consists of a lot of smaller blocks, called bundles. That OpenDaylight has this architecture, means that it is harder to get familiar with the program, but it also has some advantages. For example, not all bundles available have to be used to allow OpenDaylight to do that what is required for this project.

An additional requirement is that BECS has to be the management system in the resulting scenario. Using BECS gives some extra restrictions as well. Since BECS is software that consists of many different parts, there is not enough time to get to know the whole system. Moreover, the various parts are documented in different files and sometimes stored in diverse places. A restriction is that the BECS core may not be altered. Only the EM’s Perl scripts and Tool Command Language (TCL) scripts should be adapted for the project. There are ways to send information from the controller to BECS and use it there, but that requires knowledge and experience of different parts of BECS. Modifications would have to be made to the the BECS core to process this information. Because the BECS core may not be altered, this restricts the project to having information sent from BECS to OpenDaylight only. For error logging, information can be sent back to the EM and be stored in a log file, which has to be done.

Since SDN is flexible, traffic can go through the SDN network any way the operator of the network chooses. It is requested for this project that the packets go through the same VNFs on the way from the CPE to the gateway as on the way back from the gateway to the CPE. The only difference on the way back is that the packets have to go through the VNFs in reverse order. Section 2.4 and figure 2.7 showed that performing such SFCs is called a bi-directional service function chaining.
Also, the network created has some requirements. Since the project is not about running VNFs, using a packet bouncer that counts packets is sufficient. The hosts that are designated to execute the VNFs should be capable of running customized programs so a packet bouncer can be run on these hosts. Since [11] is written to find a proper method to perform SFC and at the same time have the network being flexible and scalable, it is required of the network to take any topology. Though the network should be adjustable considering the topology, for the test setup a limited number of nodes is used.

There is a requirement that the setup should show to the user through what VNFs the packets go. The requirement is to create a GUI with a table that shows the VNFs and the number of packets that go through each of them.

Summarizing, the requirements are:

- The OpenDaylight controller has to be used to control the network
- BECS has to be used as NMS to provide the SFCs for the CPEs
- To integrate SDN into BECS, the BECS core may not be altered
- Error messages have to be logged in case the commands BECS issues to the controller are not executed successfully
- The SFCs should be bi-directional
- The network should be able to run customized applications on the VNFs, and it should be possible to easily change topologies
- A GUI should be created that shows the user the path the packets follow through the VNFs.

3.2 Design Choices

This section shows the reasoning behind the design choices made. Various options will be evaluated in the different issues encountered.

3.2.1 Virtual Network

The first problem that arises is how to create a suitable network. As seen in chapter 1, the network has to be virtualized because there are no physical elements. There are various options on how to simulate or emulate a network with OpenFlow switches. As seen previously, a requirement on the network is that hosts in the network should be capable of running customized programs. In order to test and build the proof of concept, the VNF used will be a packet bouncer which will have to run on hosts that are designated to run VNFs. Moreover, creating different kinds of topologies is a must, and preferably it should be easy to change the topology. In a real life scenario no network emulator will be used, but for this project an open source network simulator or emulator has to be found since there is no budget for a license.

Mininet [50] was the program that was recommended by Acreo because they already had knowledge about this program. It is a network emulation program that can be used to test SDN setups. A disadvantage of Mininet is that it does not guarantee that packets are sent without any delays[51]. As
a result of this, delays and also throughput cannot be tested accurately[51]. Advantages of Mininet are that topologies and the number of end hosts can be changed by just altering a Python script, which adds a lot of flexibility. Moreover, Mininet can run any application that can be executed on the host which will be required to set up the packet bouncer.

Another application that could be used is Estinet. Estinet simulates the controller, and there are different controllers that it can simulate[51]. Next to that, with Estinet link properties can be tested accurately unlike with Mininet[51]. Estinet as a company focuses mainly on SDN and also on VNF SFC. Their network simulation tools might be well-suited for this project. However, Estinet is not an open source application, which means it cannot be used for this project as there is no budget for a license.

A couple of other network simulators are compared and discussed in [52]. A few of them (OPNET and QualNet) are not open source and cannot be used for a budgetary reason. For some of the open source network simulators, sources are explaining that SDN can be used with these simulators, like for example NS3 and OMNeT++ [53, 54, 55]. However, NS3’s OpenFlow support has not been updated since 2012, and there are some “to-do”-sections in their support page [54] which implies their OpenFlow support is not yet finished completely. OMNeT++ could be a suitable simulator since it is a very broad simulator and can use different, often used protocols. However, whether OMNeT++ can run custom-made programs on end-hosts could not be found. Another disadvantage of using either NS3 or OMNeT++ would be that there is no knowledge about these applications in Acreo, and therefore Acreo would be less capable of assisting. Two other simulators discussed in [52] are SFFNet and J-Sim. These are discarded because no information was found about SDN support.

In conclusion, it can be said that Mininet is the most suitable to use for this project. It is well-known and often used in SDN projects, and it is open source. Although it may not be very accurate on the forwarding (i.e. it cannot be guaranteed that there are no delays), it is still useful for this project as only the path packets take is important. The time packets take to reach the destination is of limited interest. Besides that, Mininet is well-known by Acreo and therefore Acreo will be more capable of supporting the project.

3.2.2 Representation of an SDN Network in BECS

In [11] the choice has been made to define three different kinds of switches, namely the CPE switch, command-and-control switch and the gateway switch. In this kind of setup, all CPEs are connected to a CPE-switch, all VNFs are connected to the command-and-control-switch, and the Internet gateway(s) are connected to the gateway switch(es). Since the different kinds of devices are connected to different switches, it is clearer to show the devices under these switches rather than just having the devices in BECS in a 1-level architecture. The controller will have to have information about where what kind of device is located, and BECS has the switches in the tree structure, so that BECS can inform the controller of the purpose of each switch. This way the controller can automatically check whether a device is a CPE, VNF or a gateway when the device comes online.

In BECS systems, the access element and underlying elements are usually shown in the tree. In case there are intermediate switches in the network which will not have any of the above described functions, a discussion can be held about whether to add those to the BECS tree or not. Adding those would increase the amount of unnecessary elements in BECS. However, it would make the network transparent to the user and in case there are any issues, BECS can provide a network overview where
the user can determine where something could be wrong. Moreover, it means that the troubleshooting data from all switches could be retrieved, rather than only from the ones to which the actual CPEs and VNFs are connected.

The controller itself will not have to be shown in the BECS tree since there is no configuration applied to it. However, the EM should have information about where to send the configuration. Getting this information in the EM can be done via a small wizard, made in TCL, in the EM’s system configuration. The system configuration wizard exists for all EMs to allow user input that is specific for that EM. For this EM, the information that will have to be provided is an IP-address and a port at which the EM can contact the controller. After that, the EM can send all information to the controller by the created socket(s).

To summarize the conclusions on how to represent an SDN network in BECS, a list of the main points is provided below.

- All switches in the SDN network will be shown in BECS
- CPEs, VNFs and the gateway(s) will be shown in BECS under the switch they are connected to
- The controller will not be shown in BECS and the EM configuration for it will be added through a wizard.

For a CPE switch, the result is shown in figure 3.1

![Design of the BECS tree](image)

Figure 3.1: Design of the BECS tree

### 3.2.3 Communication between BECS and OpenDaylight

For the communication between BECS and OpenDaylight, two solutions could be implemented. BECS already has a Simple Object Access Protocol (SOAP) API, which could be used by OpenDaylight to access BECS. However, OpenDaylight has a built-in Representational State Transfer (REST) API,
which also can be extended easily. The decision on which one is preferred, depends on whether a push or pull protocol should be used and where in BECS the API should be altered (if this API is chosen to be used).

The requirement is to have BECS send information to OpenDaylight only. Since the information sent to OpenDaylight contains configuration for a network, which is initiated by the user, the information must be processed as quickly as possible and therefore the procedure is time sensitive. For this reason, it is chosen to push this information from BECS to OpenDaylight. To implement this option, OpenDaylights' REST API has to be extended to make BECS able to communicate with OpenDaylight. In future, it is possible to use BECS' API for time-sensitive information from OpenDaylight to BECS if necessary.

3.2.4 Visualizing the SFC for the BECS user

For BECS to create a specific service chain, the user has to enter a chain in the BECS GUI. To create a way for the user to do this, there are also several choices, though all choices would involve a wizard. One choice would be to create checkboxes with services, where the user can select the checkboxes, then based on what checkboxes are selected the service chain will be created. The advantage is that as long as there are a few different services, the overview is good. However, as soon as the amount of services available becomes bigger, the overview is lost. Moreover, the order of the services is not shown if there are just checkboxes. Figure 3.2 shows this design.

![Figure 3.2: SFC wizard design with checkboxes](image)

Another option could be to have a drop-down box and an add button. When the add button is pressed, the selected service is added to a table with an index. Adding a service means that it is appended to the table. In the table, services can be selected and deleted accordingly. Since each entry in the table has an index, deleting an entry becomes more efficient. Moreover, the order in which the services are executed is shown to the user in this table. In contrary to having checkboxes, by showing the chain in a table, users can have an overview of what the service chain will be like and alter it according to their needs. Figure 3.3 shows the design with a drop-down menu and a table.
Lastly, there is a possibility to have two different lists of which one contains the available VNFs and one the VNFs chosen to be in the SFC. Besides the two lists, there are buttons to add or remove a VNF to or from the SFC. Besides that there are buttons to move individual VNFs up or down to decide on the order. Though the advantages are the same as for a drop-down box, compared to having checkboxes, the drop-down box still has two extra advantages. Since the drop-down box can have input from the user, it is possible to use this as a filter. Therefore, if there are many different available VNFs the drop-down menu can filter out ones based on user input. Another benefit is that using the up and down buttons, the BECS user can directly see and modify the order of the VNFs in the SFC. This wizard is shown in figure 3.4

The choice was made to use a drop-down menu for the selection of VNFs, because this fits with the design philosophy of BECS. Moreover, this solution gives the advantages of that it shows what the SFC is going to look like and it keeps the overview of the services up to a larger amount of services. Also, there is a possibility to filter out the results in the drop-down box easily depending on user input.

To summarize the options for SFC wizards the options and main points for each option are listed below.

- A list of checkboxes can be used, but doing so would reduce the overview for the user in case there are many different services possible to add to the SFC.
- A drop-down box could be used to select services and a table to show the current SFC. This
keeps overview in situations where there are many different services available, especially when the user is capable of filtering out services

- Two tables could be used of which one contains the available services, and one contains the services selected for the SFC.

- The drop-down box is chosen, because it is used most in BECS and provides advantages over checkboxes. It also allows the user to filter VNFs to pick from, which can be a benefit in case there are many different VNFs. Having two tables does not add significant advantages to choose it over the drop-down box.

### 3.3 Design of the Desired Model

This section takes the points of the last section and tries to create a design for a setup that matches the choices and requirements. This section derives a high-level overview of the design in the first section.

#### 3.3.1 Design Overview

To create a setup that performs everything required, different entities have to work together. There is a total of five different parts that operate and communicate with each other in order to succeed in meeting the requirements. To summarize the different entities in this project, a short list is provided below.

- **BECS**: BECS is the part that will decide on what chain is used for what CPE. It will know where to find the VNFs because the user can enter that in BECS as well.

- **OpenDaylight**: OpenDaylight issues the commands from BECS to the network and informs BECS that the commands are either issued correctly or that an error occurred.

- **Mininet**: Mininet sets up the network as required. This can be done using a Python script. It will also automatically run the VNFs on the hosts that are designated to run these.

- **MySQL Database**: To show the result to a public the VNFs have to store the amount of packets that go through them. For this, a MySQL database is used.

- **Apache web server**: The choice on how to show the result to the public, is to do this through a web page. This is readily available from almost anywhere, and if required it can even be run locally. Next to that, it allows easy access to a MySQL database.

Figure 3.5 shows how the different parts work together. In the rest of this thesis, the main focus will lay on BECS and OpenDaylight, though other parts can briefly be mentioned.
3.3.2 Detailed Desired Model

This subsection will show the reader how a more detailed figure of the setup will be implemented. The various blocks of figure 3.5 are explained in more detail and low-level figures of the blocks will be presented at the end of each of these explanations.

**BECS** - The BECS system consists of a core and one or several EMs. The core itself consists of the Central Job Manager (CJM) and the Configuration Rendering Engine (CRE). The CJM manages the BECS tree and all its objects. It takes care of so-called jobs, which are commands to change the objects in the tree from the API. The CRE gets the new objects, creates the configuration, and then changes that into instructions for the EMs. For BECS to be able to communicate with the controller, a new EM has to be written that is used to interface with it. The EMs consist of two parts. It contains of a static part written in C, which is called Element Manager Framework (EMF) and is the same for all elements, and a dynamic script written in Perl which is called Element Manager Instance (EMI). The C part communicates with the Perl part via call-back functions. The C part also communicates with the BECS core and is, therefore, the middleman in converting BECS commands into instructions for the elements. Additionally, each EM uses TCL scripts that create wizards to add user input. Figure 3.6 illustrates the overview of the internal parts of BECS.
OpenDaylight - Just a couple of the many code blocks of OpenDaylight are required for the project and to create the flows. First of all, a database of the existing hosts in the network has to be used. OpenDaylights host tracker has the capabilities to do this and for this reason, the host tracker’s functionalities are integrated into the bundle for this project. Next to that, also a list of switches and a list of links between the switches are needed to push the flows for a connection in all the switches involved. For these two demands, the switch manager and topology manager are used. These are built-in blocks in OpenDaylight as seen in section 2.4. To program the flows, OpenDaylights flow programmer is used. This only requires a flow object and the switch the flow needs to be pushed to.

The OpenDaylight core cannot communicate with the eventual northbound interface directly. For this reason, the block of code written for this project will have to have an internal API, which then can be used by a new block of code that will be the interface between OpenDaylight and BECS. The internal API will consist of functions that will be implemented by the southbound service bundle written for this project. Figure 3.7 shows the internal overview of OpenDaylight.

The northbound can be made of different types of connections. There is a built-in REST-API which could be used. OpenDaylight could also use the YANG modeling language to create the corresponding files, but that is not required. Lastly, a server socket and functionalities could be programmed all together in a new bundle.

Network - To test the calculations made, a network is required. The network in this thesis project will consist of hosts (CPEs), VNFs, a gateway, and switches. However, Mininet can only activate virtual hosts and switches. For this reason, each of the hosts in the network will have a specific functionality (namely CPE, VNF or gateway) and this has to be defined somewhere. OpenDaylight will have to be able to get this information from the place it is stored and use it to distinguish the different types of hosts. Also, to proof the hypothesis in [11], the VNFs have to be able to bounce packets back. It is not required to implement a more complex VNF.

Figure 3.7: OpenDaylight Internal Overview
Web and MySQL Server - Lastly, the entity that shows the results has to be made. In this case this will consist of a MySQL server and a web server. The database in the MySQL server will contain the different hosts that are VNFs and the amount of packets that go through those VNFs. The web server contains some PHP: Hypertext Preprocessor (PHP) files that will create the HyperText Mark-up Language (HTML) layout. It will also send a JavaScript to run on the client, which will fetch the database information through Asynchronous JavaScript and XML (AJAX) from a PHP script, which in itself fetches the data through the Structured Query Language (SQL) from the database. The JavaScript will also create the table to show the information in the database to the user. Figure 3.8 shows the different blocks of the desired displaying model.

3.3.3 Summary

Previously in section 3.3, the different entities were explained in detail. This section provides a summary of the details discussed in that section, and it also combines the information to get a detailed overall overview. The complete overview is shown in figure 3.9

As seen previously in this chapter, the user will use BECS to choose SFCs for the CPEs. For this, a new EM has to be developed. This EM has TCL scripts that allow the user to input SFCs, which can be added to the CPEs. CRE will render the user input configuration for the different CPEs and send the instructions to the EMF. The EMF in its turn sends the information to the EMI spawn, which processes the information to match the protocol used between BECS and OpenDaylight.

Within OpenDaylight three different entities have to be written: a southbound bundle, a northbound bundle, and an API that can be used between these. The northbound bundle is an extension to the already existing REST API and is solely responsible for interfacing BECS with the southbound bundle. However, since the northbound bundle cannot call southbound bundle functions directly, an API has to be developed that interfaces these two. The southbound bundle is the part that calculates the path between CPE and gateway. To do this, it requires an instance of OpenDaylights host tracker and switch manager. The southbound bundle uses OpenFlow to communicate the calculated path to the switches in the network.
Mininet will be used as the network emulator and will create the network via a Python script. Each of the VNF hosts will run a packet bouncer as VNF. The packet bouncers store the number of packets they bounce in an SQL database. An Apache web server uses PHP to retrieve the information from the database and to display it to the user.

Figure 3.9: Detailed overview
4. Integrating SDN in Third-Party Network Management Software

As seen in chapter 1, the BECS system is currently unable to communicate with SDN networks since it communicates with the control plane of network elements. For this reason, the controller and the NMS BECS will have to be extended to make BECS able to perform service provisioning in SDN networks through OpenDaylight. Chapter 3 showed the choices made and the design that was created based on the different choices. This chapter explains how the design was implemented, what issues were found with the design and how those issues have been solved. Moreover, at the end of the chapter, a discussion is held about some parts of the design that could be improved.

4.1 The Chosen Topology

This section provides a brief overview of the topologies that were available and which one has been chosen. The details about the choices made are discussed in [11], but to give a complete picture, background information about them is presented in this section.

The first topology consists of a single switch to which all the CPEs, VNFs, and gateways are connected. This topology has the disadvantage of all the rules being placed in one switch. The topology will be simple, but there is no scalability, redundancy or flexibility. Figure 4.1 shows what this concept would look like.

![Diagram of the first concept for network topology](image)

**Figure 4.1: First Concept for Network Topology**

To introduce more scalability, two new topologies are found. In one concept the CPEs are connected to a switch that forms the start of a tunnel. This switch forwards traffic to the next switch, which is connected to a VNF and another switch. The switch checks whether a packet should go
through the VNF or not. If so, it will send the packet to the VNF. Otherwise, it will send it to the next switch. This procedure continues until the packet arrives at the gateway. On the way back, this happens in reverse order. The advantage is that one can add as many VNFs as is desired and that the ways to and from the gateway go through a bi-directional service chain automatically. However, the topology is not flexible at all, because VNFs have to be used in a fixed order. Redundancy can be achieved by adding an extra switch, which is connected to all the other switches and will bypass switches or connections that are down. Figure 4.2a illustrates this concept.

The second improved concept, which is shown in figure 4.2b, has switches with different purposes. There are client switches, which are only connected to CPEs and have an external port to the so-called command-and-control switch. The command-and-control switch is connected to the VNFs and all other switches. Lastly, there is a gateway switch, which is connected to the Internet gateway(s). This topology has advantages of scalability; one can add multiple client switches to provide services for more customers. In case there are too many VNFs or redundancy has to be added, a second command-and-control switch could be connected to the network. Also, the flexibility of traffic routing is large in the sense that one can choose whichever order of VNFs the traffic should go through. Next to that, this set-up could also grant host classification easily, since the classification could be made on the kind of switch a host is connected to.

In both improved concepts, one can group packets by adding a VLAN tag at the client switch, which highly reduces the amount of rules set in the switches. However, doing this introduces a new problem. The VNFs might not be located in the same place, and this means that there might be at least one router in the middle, which would remove the VLAN tag because VLAN tags are placed inside the ethernet header.

To solve this problem, an additional solution has been found. Figure 4.3 explains this concept graphically. Each client switch is connected to a tunnel gateway, which will encapsulate the packet including VLAN tag. When it arrives at the command-and-control side, the other end of the tunnel removes the encapsulation so the command-and-control switch can handle the packet. Then for each VNF side the packet needs to go to, the packet has to be encapsulated again and decapsulated at the VNF side. After going through the VNF(s) this has to be done again for the way back to the command-and-control switch. The command-and-control switch then sends the packet to the gateway. On the way back the command-and-control switch addresses the VNFs in reverse order.
The study in [11] shows that the solution with the different kinds of switches is best-suited. Therefore, the solution shown in figure 4.2b is chosen to be used for the project. This because the various client switches can grant flexibility, scalability, and redundancy.

4.2 Implementation

This section discusses how the proposed design, shown in chapter 3, has been implemented. It will go into details for each part of the design.

4.2.1 Creating the Network

After the topology is chosen, the network can be set up. As seen in chapter 3, for this Mininet has been selected. Since a customized topology is required as seen in the previous section, a Python script has to be written to automate the set-up of the network. Mininet only has some standard topologies and the chosen topology is not one of these. As seen in section 4.1, the chosen topology has three kinds of switches; the client switch, the command-and-control switch, and the gateway switch. Switches should not be overloaded by rules if there are many CPEs in the network and therefore the number of client switches has to be able to be increased, which means the network created by the Python script has to be extendable.

To prove that packets go through the right VNFs, each of the VNFs has to have a program running to count the packets. Moreover, the VNFs are required to send the packets back from where they came. To do so, a packet bouncer is written. The bouncer is written in C and at first, it is made capable of just bouncing packets. This is done since it is required that packets be bounced back to test whether the future rules are correctly installed. Later on in this chapter an extension to the bouncer will be explained. In the Python script written to create the topology, it is possible to add a CLI command for specific hosts. The bouncer is run on the hosts that are designated to be a VNF using these CLI commands.

While setting up the network, it was found that, by default, Mininet searches for a network controller on the loopback interface of the machine it is running on. Since OpenDaylight is not running on the same machine, also the remote controller has to be specified for Mininet and OpenDaylight to be able to communicate. This has also been added to the Python script.
4.2.2 Implementing the SDN Controller’s Southbound Interface

In chapter 3, it was shown that the controller in this project is divided into three parts; the northbound bundle, the southbound bundle and an API between these two. This section explains how the southbound bundle is implemented. Chapter 3 also showed that different existing parts of OpenDaylight have to be used; namely, the host tracker, flow programmer, topology manager, and switch manager.

After understanding what kind of variables are required to create a flow (a set of matches and actions), the flow rules can be created. The rules that have been found earlier to create service chaining of VNFs were first statically programmed into the switches by OpenDaylight. These rules are shown below.

1. Incoming packet from CPE should go to the first VNF
2. As long as the packet has to go through more VNFs, send it to the next one
3. When the packet comes from the last VNF, send it to the gateway.

On the way back the incoming packet will come from the gateway and in the end it will be forwarded to the CPE. At first, only the most simple solution as shown in figure 4.1 is used, because then the flows only need to be programmed in one switch. When the OpenFlow rules are installed in one switch successfully, the program can be extended to install them in multiple switches at the same time to create one flow.

When testing whether this works, it was found that, due to bouncing the packets the MAC and IP addresses are changed (shown in figure 4.4). The IP addresses could be changed within the bouncer, but the MAC addresses could not. Consequently, the gateway will not receive the packets. For this reason, packets coming from a VNF were required to be altered as well as forwarded. Some changes are made to the controller to make the switches alter the MAC and IP headers as well. These changes are made based on what VNF is known by the controller to be on that port. After that, the actual set up can be made as shown in figure 4.2b earlier.

![Figure 4.4: Overview of Data Link Layer and Network Layer Addresses](image-url)
The topology that has been chosen, as shown in figure 4.2b, introduces a new problem. When having multiple switches in the network, OpenDaylights host tracker in the first version of the controller did not show the hosts correctly. It noticed hosts on the wrong switches and also showed the wrong topology on the GUI. Printing all properties of each host confirmed this error. Since the host tracker is required to find what host is on which port, a custom host tracker was made. The below paragraphs explain how the information needed to perform the SFC is gained, stored and used.

A host object was created within the Java code to manipulate a host’s data easily. It can be extended to a VNF or CPE and contains addresses and physical location of the host (i.e. which port on what switch the host is connected to). In the case of a VNF, the type of the VNF can be added. In the case of a CPE, an SFC can be added. The variables for each host are saved in the host object as soon as a packet from a host arrives at OpenDaylight. The decision whether a host is a client, a VNF or a gateway is taken by looking at the switch it is connected to. However, this adds some static variables to define the type of the switches in the code, which has to be taken away and can be set via BECS at a later stage of the implementation.

Because there are multiple switches in this scenario, the ports between the switches have to be defined as well to create a connection between CPE and gateway. As soon as the first packet arrives, the topology manager will be accessed to create a switch list. The switch list contains objects of the type switch, which have an ID and a list of links to other switches. This way the switch objects and switch list can automatically be updated when new switches enter the network or if links go down.

Since the hosts are now found on the right port using the newly created host tracker, the rules can be created. At the client switch, the external port (port to the command-and-control switch) is found first, using the switch list described above. All packets coming from this port will be put on the output port of the concerned CPE. All other packets will be sent out on the external port.

At the gateway switch, the same was done. However, only one gateway will be taken into account and therefore the rules can be more simplified. Every packet that has the gateway as destination will be sent to the gateway. All other traffic is forwarded via the external port to the command-and-control switch.

The command-and-control switch itself has more complicated rules and is shown below. Remember that the headers have to be modified due to the packet bouncing. On the way back the incoming packet comes from the gateway switch and is sent to the client switch after going through the VNFs in reverse order, like a bi-directional SFC. The rules are enumerated below.

1. Incoming packet from a client switch has to go to the first VNF
2. As long as the packet has to go through more VNFs, find the correct MAC and IP addresses, change the corresponding headers, and send it to the next VNF
3. When the packet comes from the last VNF, send it to the gateway switch.

These rules finalize the first version of the southbound interface of the controller. The next section will talk about the northbound interface.
4.2.3 Programming the SDN Controller’s Northbound Interface

For the northbound interface, a couple of choices have to be made as well. The first decision to make is which of the two programs would be the server and which the client. In chapter 3 it was decided that OpenDaylight should be the server in the communication with BECS since its REST-API was chosen to be used. The second choice to make is what the different requests BECS has to make will be and how to design them.

To create the REST-API a new bundle has to be created, because there is a class that is used as bundle activator and these bundles are different classes for northbound and southbound. This bundle can then use the southbound service API to use the functionalities in southbound service.

The part of the URL that will define that it is a command to the REST-API has to be chosen. This part of the URL, also called the base URL, is followed by the name of the function that is called, which is followed again by the variables that this function requires. The path that is general for all BECS commands is defined in an external file. The path to the functions was defined in the Java code.

Also, the type of request needs to be defined. As by definition HTTP has different types of requests, among which the GET, POST, DELETE and PUT requests [56]. The DELETE and GET are used to delete and to read a resource respectively in REST[57]. In REST-APIs the PUT and POST methods can both be used to alter resources[57]. POST requests are often used for updates and PUT requests to add new resources. Since the SFC changes are not sent, but only the SFC as a whole, the PUT-request was chosen in this scenario.

To meet the requirements and get all the information necessary in OpenDaylight, three different API calls have been defined. First of all, due to the way the topology is set up in [11] and that OpenDaylight has to determine what kind of host a packet comes from, BECS should be able to send information about what type a switch is (e.g. a CPE switch). Secondly, to create SFCs, the VNFs have to have a function. The previous section showed that a host object could be extended to a VNF, for which the type can be added. OpenDaylight automatically notices the VNFs, but assigns a type based on what information BECS sends about the VNF. So there has to be a call that can specify the type of a VNF. Thirdly, the chain has to be specified for a specific CPE. Similarly to a VNF, the CPE object is extended to include an SFC.

To summarize the calls that have been implemented into the northbound interface the below list is provided:

- set_switch_type: To allow BECS to set the type of a switch in OpenDaylight (e.g. CPE switch)
- set_VNF_type: To allow BECS to set the type of a VNF (e.g. firewall)
- set_SFC_chain: To allow BECS to set an SFC for a CPE (e.g. firewall -> DPI)

4.2.4 Southbound Service API

The southbound bundle is called a service because it provides the functions to communicate with the network. The northbound and the southbound are both implemented at this point, and it is understood from the use of previous packages (that already existed in OpenDaylight) that the northbound could call functions from the southbound service directly. However, when trying to call functions from the southbound service directly from the northbound REST-API, the functions are not called. The HTTP client used to test the southbound service and northbound API is Postman[58]. When looking more deeply into how the bundles are built, it was noticed that all service bundles have an API and
the southbound service bundle will also require one to be used by the northbound. First, a short enumeration of the API functions will be provided, after which a more detailed explanation will be given.

1. set_SFC_chain(cpeIPaddr, vnfArray) - To set the SFC for a CPE, the IP address of that CPE is sent along with an array of VNF services it requires

2. set_VNF_type(nfvIPaddr, vnfType) - To set the type of a VNF, the IP address of the VNF is sent along with the type of service that the VNF offers

3. set_switch_type(switchId, switchType) - To set the type of a switch, the switch ID is sent together with the type of that switch

4. get_err_msg() - To get the error message, one of the previous calls has to return false. Then the error message can be retrieved.

set_SFC_chain - The first function to implement is to make it possible to alter services for a specific CPE, but how to define the CPE and the VNFs through which the traffic has to go? The CPE would be defined by IP address as this is commonly done on the Internet as well as in BECS services. However, the VNFs could be defined by IP address or by the type of service they offer. Since BECS uses a variable that is readable by the users to define a service chain or VNF type, the choice is made that the service chain will be defined by the types of the VNFs (e.g. firewall or NAT), and not the IP addresses. To be able to add multiple services for one CPE, the service types are divided by a specific character.

set_VNF_type - In the previous tests, the VNFs were hard coded since there was no way to give the input to OpenDaylight yet and the different CPEs had different service chains. However, there has to be some entity that provides information about what machine offers which service(s). Since BECS has information about all the hosts in the network and to what switch they are connected, it is assumed that BECS has information about where a VNF can be found. Therefore, the second function that had to be written in the API was a function that defines the service a VNF offers. Note that OpenDaylight has stored information about what hosts in the network are VNFs, since that can be defined by the switch they are connected to. In conclusion, to define the VNF for which the service has to be set, the IP address is used. To define the service the host offers, the name of the service is used. The name has to be used, because users will have to be able to read or alter the service type of a VNF.

set_switch_type - The next function that is implemented is the function to define what switch is of what type. As has been explained earlier, there are three kinds of switches; client switches, command-and-control switches, and gateway switches. OpenDaylight will be aware of what switch is of what type to classify the hosts. Since the topology of the network may vary in future, this cannot be hard coded inside the program. Because BECS holds all information about the switches in the network, a variable can be added to those switches which classifies them. This variable can be either client, command-and-control or gateway. It makes the classification of switches readable for the user. The switches are identified by the switch ID, which is an integer defined by OpenDaylight. Each switch has the same ID in BECS. So this function gives the switch ID and the type of the switch to OpenDaylight. After this OpenDaylight can classify the hosts.
Lastly, it was required that the server would return a 200 OK message to BECS if the call was made successfully or an error message in case an error occurred. The southbound service has a global variable that will change when an error occurs, having both an error message ID and an error message text. The API contains a function that retrieves this variable, so it becomes available at the northbound. This way the northbound can send the error or, in case the error variable is undefined, a 200 OK message back to BECS.

4.2.5 Programming BECS’ Element Manager

The last step is to make BECS capable of calling the REST-API of OpenDaylight. As seen before, each different kind of element that BECS manages requires a new EM. So OpenDaylight would require such an EM as well. As previously stated an EM consists of two parts; the static part written in C and the dynamic part written in Perl. Next to that a set of TCL scripts is added to create wizards. The part written in C calls call-back functions from the Perl part. The C part does not have to be touched, only the callbacks will have to be changed to make BECS capable of communicating with OpenDaylight. To allow the creation of an SFC, at least one TCL script has to be altered.

In order to create the configuration, it is required to be able to create an SFC. To create a service for the SFC a service definition has to be created. This is done using the already existing service definition wizard. This wizard provides the user with several choices, e.g. whether it should be a service with static address, dynamic address or without an address. For the service chain for a specific CPE, an address is required so BECS can define what CPE should get the SFC. Since the addresses are statically defined in Mininet, also a static address is used for the service in BECS. A new option is added to what kind of service it is. Since BECS can provision many different elements, there are many different kinds of services, like Virtual Private Network (VPN) functions, CPE functions, and Multiprotocol Label Switching (MPLS) functions. A new kind of function is added here called VNF because the function that is created will be for VNFs. This function has a mandatory field that contains the SFC, which is created when the service definition is added (or mapped) to a specific interface on the OpenFlow switch. The creation of the SFC is then done in another wizard because not all interfaces may have the same SFC.

When mapping the service definition to an interface, the second wizard will provide a user interface to select the wanted SFC. This wizard is developed in TCL and is called using already existing service mapping procedures. The service mapping procedures first request what IP address the service should have. In this case, this is the address assigned to the CPE. Since the service definition contains the service type VNF, the service mapping procedure can distinguish the wizard that has to be called for the service mapping after the IP address is chosen. As discussed in chapter 3, the wizard created has a drop-down menu to add VNFs to the SFC in a table. The table shows the order of which the different VNFs are used. The wizard is shown in 4.5. The result in the BECS tree after mapping the service to the interface is shown in 4.6.
Chapter 4. Integrating SDN in Third-Party Network Management Software

After the SFC has been created, it is possible to send configuration down to the EMI. Configuration is grouped in contexts. For these contexts, first a "context open" is called. This registers the context to the EMI for a specific element. After that, the EMF calls a "context write". Within the context write, the configuration is sent and the EM can process the configuration. In the case of this EM, it is required to translate the configuration into the REST-API requests. For this project, a separate Perl file is created which performs this translation. This Perl file is then included in the main perl file for the EM.

The EM uses internal commands inside the configuration, which are lines that start with "#em emi". These lines have to be skipped while parsing configuration for OpenDaylight, but can be used to set specific global variables, like OpenDaylight’s server IP and port. Configuration lines that are created by the EM look like below.

- client=<IP address> vnf=<vnf type> : to set the VNF type for a host designated to be a VNF
- client=<IP address> vnf=<sfc chain> : to set the SFC for a specific client.
- odl-role=<role> : To set the role of an OpenFlow switch
- odl-ofid=<ID> : Sends the OpenFlow switch ID as configuration, so the EM can use it.

Based on the above configuration lines and the global configuration lines containing the IP address and port of OpenDaylight’s server, the HTTP requests can be built. Perl has default HTTP libraries
that can act as HTTP client. The LWP library is imported to create HTTP requests and retrieve the answers. HTTP’s ”keep-alive” option is used to prevent the TCP connection from being torn down after every call, because in future much more data may have to be sent since a network could consist of thousands of elements. This reduces the amount of traffic in case of failure, start up or other situations where much configuration data has to be sent to the network. The response to the requests can be either a 200 OK message or an error message with an error code. To provide the possibility to see what went wrong in case of an error, a function to write to a log file has been created. This file contains time stamps and the error code and message.

4.2.6 Displaying Results

As seen before, to display the way the resulting packets go, a real-time web page has been created on a web server. The web page communicates with a MySQL database. The MySQL server is set up, which contains a database with the VNFs and the amount of packets that are sent through them. However, the bouncer was only bouncing packets. For this reason, it was extended to send the count directly to the database, through SQL ”insert into” and ”update”-statements. However, the only variable that can be sent was the interface name. Interface names are not easy to read for outsiders or users and that is why a mapping has to be created. The question is, where can this mapping be made? The interface name contains the name of the host in Mininet (e.g. n1 for VNF 1) and the port. However, neither OpenDaylight nor BECS receives information about what names the VNFs in Mininet have.

There is a systematicness in how Mininet chooses the names, however, due to the Python script. The lowest one (n1) will always have the lowest IP address in the network. The OpenDaylight controller is being executed on the same server as the web server. The BECS server is located in a different machine. For this reason, it is logical to have OpenDaylight create a text file that contains a mapping and the PHP from the web server can then read this file and replace the interface names from the MySQL database with the service that is provided by the VNF on that interface. Figure 4.7 shows the flow chart of the communication between BECS, OpenDaylight, and the network. The VNF_Object_Array is inside OpenDaylight. Figure 4.8 shows the final environment in a bit more detail.

![Figure 4.7: Systems’ Flow Chart](image)
4.3 Implementation Discussion

This section discusses implementation choices. Some advantages and disadvantages and advice are provided as well.

4.3.1 Controller

Although the controller to use has to be OpenDaylight (this is a requirement), it may not be the best suitable controller for the project. As commonly known among network engineers, one of the disadvantages of SDN is the fact that the centralized control plane is a single point of failure. While in legacy networks if a router’s control plane fails only the router fails in that network, in SDN the whole network controlled by the controller will fail. At least new rules will not be pushed to the network. Besides that OpenDaylight is slow on start-up (several minutes), so using it during the project will require much more time. Also, OpenDaylight is written in Java, which is known to use substantially more memory and more of the Central Processing Unit (CPU) than some other languages\[59\].

However, the biggest issue still is the single point of failure of an SDN controller. Though NOX and POX are not as complex and faster on start-up than OpenDaylight, their internal structure is quite similar. There is one instance of the controller which manages the network. A better way to have a controller would be to distribute it, like for example ONOS does. While it is possible to write modules for NOX, POX, and OpenDaylight to make these controllers distributed, these kinds of modules are not there by default. ONOS can have multiple instances that will each control a different part of the network. If one of the instances goes down, then the rest of the instances will take over the switches that were controlled by the first instance. This way the single point of failure is reduced significantly. Unfortunately, ONOS instances need to be synchronized and to do so they share a NIB. If there is a failure when ONOS controller instances share the NIB, then this could still be a single point of failure. Moreover, the NIB can become large\[44\], which may be hard to synchronize among all controller instances.

The latest development is that of the DISCO controllers. The instances of these controllers manage a part of the network and communicate for end-to-end communication using a lightweight protocol. By doing this, DISCO does not need the NIB that ONOS uses or another large data-structure that
has to be synchronized between the different controller instances. However, even though DISCO controller instances will adapt the routing between their domains if links go down, it is nowhere stated whether the switches will be taken over by other instances. If this is not properly addressed, parts of the network could become unmanaged, whereas ONOS instances would work together to manage all of the network elements.

All in all, each of the above-described solutions has advantages and disadvantages. However, even though OpenDaylight has to be used for the project, it is far from the best solution possible as the results of it failing could be catastrophic. Depending on whether the importance lies in having all switches connected to a controller instance or reducing the amount of data transferred between controller instances, either ONOS or DISCO would be better solutions.

### 4.3.2 Change Test Bouncer to a Real VNF

The next step is setting up a network that contains real VNFs. In the proof of concept setup, a bouncer was used that just bounces the packets. However, in real networks, a VNF will not be a bouncer. VNFs could be anything and will in some cases change the packet headers, like in the cases of a NAT or proxy. To be capable of handling this kind of traffic flows, the controller will need to be altered. The current way the flows are set will not suffice since it cannot add VNF-based traffic flows.

![Network Using a NAT or Proxy](image)

**Figure 4.9: Network Using a NAT or Proxy**

<table>
<thead>
<tr>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source IP: 70.120.0.1 &amp; In Port:1</td>
<td>Out Port: 2</td>
</tr>
<tr>
<td>Source IP: 70.120.0.1 &amp; In Port:2</td>
<td>Reset Packet headers &amp; Out Port: 3</td>
</tr>
<tr>
<td>Destination IP: 70.120.0.1 &amp; In Port:3</td>
<td>Out Port: 2</td>
</tr>
<tr>
<td>Destination IP: 70.120.0.1 &amp; In Port:2</td>
<td>Reset Packet headers &amp; Out Port: 1</td>
</tr>
</tbody>
</table>

The switches currently only look at source and destination IP addresses. When a proxy or NAT is somewhere in between, the controller will have to create flows based on the IP address and also the transport layer port that the VNF provides on the external side. The new flows are required, because the NAT or proxy will change these fields in the headers. Figure 4.9 shows what will happen
and table 4.1 shows the flows that would be currently set in the switch. Please note that OpenFlow switches can also check and alter network and transport layer headers and are therefore capable of linking different networks. Reset the packet headers means finding the MAC addresses based on the IP addresses. Since the NAT or proxy should change the network layer header, the packet will not match any of the flows in the switch table whereas it should match the flow shown in red.

There is a way to create a solution to this issue. BECS can connect to a VNF like a NAT since it will have an external port. In the case of a static IP NAT, BECS can then configure the NAT to use specific IP addresses and ports. This information can be communicated to OpenDaylight and OpenDaylight can then alter the rules in the OpenFlow switch accordingly. When the VNF uses dynamic IP NAT, this may not work, since the IP address is chosen from a pool. Another solution would be for BECS to retrieve the configuration from an element that alters the network layer header of packets. When BECS has retrieved the configuration, it can be sent to the controller, which in turn can change the rules in the switches.

4.3.3 Allow Moving Hosts and Multiple Hosts on One Interface

This section describes two situations, which could occur and OpenDaylight’s southbound bundle should take into account.

In real commercial networks with CPEs sometimes a CPE breaks. In this case, it should be possible to replace the CPEs, which would mean that the MAC address of the device changes. Since BECS handles the leasing of IP addresses, it does not mean that the IP address changes as well. Such a situation where the CPE is replaced should not disturb the provisioning of services and therefore the southbound bundle should be able to handle such a situation.

There are also different scenarios in which multiple CPEs or WAN interfaces could be linked to the same port. For example, in general, apartment buildings have a common switch to which all apartments are connected. The switch is then connected to an access element. Another example could be a user that has a couple of Voice over IP (VoIP) phones, IPTVs, and computers. Each of these devices could potentially use a separate CPE or WAN interface. The result is that the southbound bundle should be able to allow multiple IP addresses on the same port of a switch.

To support the above situations, the choice was made to use the MAC address as the unique identifier of a host. When the host would move, the southbound bundle would update the switch and port accordingly. By using the MAC address as the unique identifier, it is also possible to have multiple CPEs or WAN connections on the same interface of a switch. Each device or WAN connection would get an IP address and therefore a specific set of rules in the CPE-switch.

Even though the code would support these situations, no way was found to test the above scenarios in Mininet. Therefore, it was not possible to test these specific scenarios.

4.3.4 Implementation Discussion Summary

In this section, different parts of the implementation that could be improved have been discussed. This section tries to highlight the most important points from the discussion, which are shown below.
• The OpenDaylight controller is a single instance controller which means that if it fails, the network could temporarily have no controller. This situation could result in the wrong configuration being applied to the network or configuration could time out. A solution could be to use either the ONOS controller or DISCO concept. ONOS uses multiple controller instances which could take over failing instances, but shares data which could become large among the instances. DISCO uses a lightweight protocol between the controller instances, but those controller instances do not take over if another instance fails.

• During the project, only a packet bouncer was used as VNF. The packet bouncer does not change the packet itself, which is something real VNFs could do. The controller will have to address these changes, which means that the rules applied to the network in this project are not sufficient anymore. However, to adapt the controller to set rules according to what is needed, it may be required of BECS to receive information about what the VNF does with packets. This implies the northbound API of OpenDaylight might have to be extended, and a way has to be found for BECS to receive information about how packets are altered.

• Though moving devices and having multiple devices at the same port of a switch should be supported using the southbound bundle created, there was no way found to test this.
5. Analysis

In this chapter, the analysis is performed. First, a discussion will be held about what is tried to achieve. After that, the data retrieved from the set-up is provided to the user. Lastly, conclusions will be drawn regarding the collected data. The network set-up used to analyze the data is shown in figure 5.1. The complete environment including the management is shown in figure 4.8.

5.1 Analysis Goals

Before any analysis can be performed, first the analysis goals need to be defined. What is it that should be analyzed and why?

It is not self-evident that the set-up works for different commonly used situations by ISPs. For example because of multi-tenancy traffic might get mixed up. The analysis tries to explore what common situations for ISPs the system can deal with, using the web page as a tool for displaying the results in the network. Unfortunately, replacing a CPE and having multiple devices on the same interface could not be analyzed due to limitations in the network emulator as explained earlier. The situations that will be taken into account are shown below.

- Due to scalability, there may be multiple CPE switches to support a bigger number of CPEs.
- There will be multiple CPEs on the same switch, which has to be correctly identified by OpenDaylight.
The first goal of the analysis is to explore the above situations and compare the resulting traffic with the SFCs for different CPEs, without having interference between the CPEs.

In this thesis, a set-up was created to perform SFC using BECS and SDN. However, when BECS has configured the controller it does not automatically result in the correct configuration. There is human interaction when creating the configuration for OpenDaylight in BECS, which could be wrong. This analysis’ second goal is to investigate whether the web page developed helps in finding network configuration errors and network failures. If the web page can help to do so, it can save a significant amount of time when an ISP gets complaints from customers. In such case, it could be used as a troubleshooting tool.

5.2 Data Analyses on the Network

This section provides the reader with information about the analyses of the data that goes through the network. A web page is developed to display the data that is collected. First, the set-up regarding CPEs on the same switch is discussed. The section after that, also CPEs on the two different switches will be considered. During this section, the network setup as shown in figure 5.1 is used.

5.2.1 Data Analysis on the Network Considering One Client Switch

This section explores different situations in the network, where the CPEs are on the same switch. The complexity of the situations will be built up during the section.

The CPE with IP address 10.0.0.3 has been given two services; firewall and DPI. Mininet gives the option to ping from one host to another, but also to send an HTTP request. If the CPE pings the gateway, the packets should go through both the firewall and DPI. Figure 5.2 shows what happens.

The service chain that has been set is:

1. CPE$_{10.0.0.3}$ -> Firewall -> DPI -> Gateway
2. Gateway -> DPI -> Firewall -> CPE$_{10.0.0.3}$

![Figure 5.2: Resulting Packets through Firewall and DPI with One CPE](image-url)
The green background in figure 5.2 indicates that the value has been changed in the past second. The values of the packet counters increase by two at a time however. The values of the counters increase by two each time because the packets go through the VNFs on the way to and from the gateway. The fact that the values on the web site change shows that the flows have been set and are correct. These results apply to having only one CPE on one switch. The next environment to test would be to have two different CPEs on the same switch. For this reason, another CPE, with IP address 10.0.0.4 will receive the service NAT. The result of this is shown in figure 5.3. Also, the CPE with address 10.0.0.3 will only have a firewall service. Please note that the values of the packet counters have been reset. The following enumeration shows the chains in more detail:

1. CPE_{10.0.0.3} -> Firewall -> Gateway
2. Gateway -> Firewall -> CPE_{10.0.0.3}
3. CPE_{10.0.0.4} -> NAT -> Gateway
4. Gateway -> NAT -> CPE_{10.0.0.4}

As can be seen in figure 5.3 both the firewall and NAT are receiving packets. First, the service chain with (the one for 10.0.0.3) is activated and goes through the firewall. The number of packets that went through the firewall is now increasing. After that, the second service chain is activated to go through the NAT. From this point, the number of packets that went through the NAT is now increasing as well. The difference in the amount of packets is due to the delay between setting the first SFC and setting the second SFC. The result here shows that the packets that are sent from two different locations are also passed through the VNFs. The packets also arrive at both destinations, which means on the way back the packets are parsed correctly as well. To provide multi-tenancy, the next step is to investigate if the packets from different CPEs can go through the same VNF too. In order to do this, the CPE with address 10.0.0.3 will receive a new service in addition to the firewall it already has, namely the same NAT as through which packets from 10.0.0.4 go. The result is shown in figure 5.4 and in the following enumeration:
Chapter 5. Analysis

1. CPE\textsubscript{10.0.0.3} -> Firewall -> NAT -> Gateway
2. Gateway -> NAT -> Firewall -> CPE\textsubscript{10.0.0.3}
3. CPE\textsubscript{10.0.0.4} -> NAT -> Gateway
4. Gateway -> NAT -> CPE\textsubscript{10.0.0.4}

Figure 5.4: Resulting Packets through Firewall and NAT having Multi-Tenancy

The value for the NAT in figure 5.4 is much higher now than the value for firewall. The reason for this result is that both the CPEs send their packets through the NAT, whereas only one sends its packets through the firewall. The increase of packets per step is now four instead of two. At both the CPEs the packets arrive and therefore the result is that the flows are correctly set and that both traffic flows go through the NAT. It would be expected that the number of packets in the NAT is exactly two times higher than the number of packets through the firewall, which is not the case. This can be explained by the delay between creating the first and second chain.

5.2.2 Data Analysis on the Network Considering Multiple Client Switches

This section contains the data analysis on the network where multiple CPEs are divided over two different client switches, namely, switches one and four in figure 5.1. This set-up adds some extra complexity in the command-and-control switch as this one has to decide to what client switch to send the packets to on the way back. This decision is made by looking up to what switch the client is connected by using the IP address.

The NAT is removed from the service chain of the CPE with address 10.0.0.3. Thus this CPE only keeps the firewall service. On the second client switch, a CPE with address 10.0.0.6 is set to have a load balancer in the service chain now. The result should be that two different traffic flows from two different client switches are forwarded correctly as well. The values of the database have been reset again to make the result clearer. Figure 5.5 shows the result of this test. The service chains are shown below:
Chapter 5. Analysis

1. CPE_{10.0.0.3} -> Firewall -> Gateway
2. Gateway -> Firewall -> CPE_{10.0.0.3}
3. CPE_{10.0.0.6} -> Load Balancer -> Gateway
4. Gateway -> Load Balancer -> CPE_{10.0.0.6}

Figure 5.5: Resulting Packets considering Two CPEs on Different Client Switches

The above figure clearly shows the same result as figure 5.3 with the only difference being that here the CPEs are located on different client switches and the CPE with address 10.0.0.6 has a different service. The difference in amounts of packets is again the result of the delay between creating the SFCs. The last situation to consider is that multi-tenancy can be provided among CPEs located on different client switches. To test this situation, the CPE with address 10.0.0.6 also sends its traffic through a firewall. Figure 5.6 shows the result of this. The service chains for both CPEs in both directions are shown below:

1. CPE_{10.0.0.3} -> Firewall -> Gateway
2. Gateway -> Firewall -> CPE_{10.0.0.3}
3. CPE_{10.0.0.6} -> Load Balancer -> Firewall -> Gateway
4. Gateway -> Firewall -> Load Balancer -> CPE_{10.0.0.6}

Figure 5.6: Resulting Packets considering Multi-Tenancy for Two CPEs on Different Client Switches
Chapter 5. Analysis

The amount of packets that go through the firewall is approximately twice as big as the amount of packets that go through the load balancer. This result is as expected as the packets of both traffic flows go through the firewall, which is the same result shown in figure 5.4. Since the packets arrive at both the CPEs on the way back, the packets have to have taken the right links in the network.

5.2.3 Network Troubleshooting

In general, it can be quite hard to perform network troubleshooting. This section gives a short analysis on how BECS in combination with the Web GUI presented earlier can make it easier for the operator to perform network troubleshooting.

When a customer has a complaint because he does not have a connection to the Internet, a couple of things can be wrong considering SFCs that prevent packets from reaching the Internet. First of all, one of the services could not be active at the moment for some reason. The result is that the service does not send the packets further and therefore they are dropped. Another issue that could occur is that the BECS user enters an invalid service in the SFC. Of course, the services available are pre-defined in the wizard used to create an SFC. However, the SFCs are stored in a parameter at some point, and the BECS user would be free to alter this parameter manually. Lastly, it could be that one of the switches is down.

The first situation that is explored is that the user input is erroneous. To keep it simple a mistake was made on purpose in the firewall service, and it was written as "firewall". Figure 5.7 shows the result of this configuration error in BECS. BECS receives an error message from OpenDaylight and displays to the user that the input is invalid. When the user sees this error, he or she can immediately see that there is an error in the configuration and alter it accordingly without having to check the network.

Besides the possibility of human errors like a simple typing error, it is possible that there are issues with the network itself. Normally, if traffic is sent out but no response is received, the network operator has to check different elements in the network and their interfaces to find what is wrong. A commonly used tool for this is Tcpdump or a debugging command on the element itself. Using these tools, often on different elements, is cumbersome and it often takes time to find the issue. To re-create such a situation, consider the below SFC:

1. CPE$_{10.0.0.3}$ -> Firewall -> NAT -> Gateway
2. Gateway -> NAT -> Firewall -> CPE$_{10.0.0.3}$

However, for some reason, the NAT service is down and therefore does not forward the traffic. This is simulated by not running the packet bouncer on the server that should run the NAT. Because the traffic is not forwarded, the user at 10.0.0.3 sees that the Internet connection is down. Since the
user’s CPE is still sending out traffic, the network operator can see that the traffic passes through the firewall (as shown in figure 5.8), using the web GUI. On the other hand, no traffic goes through the NAT. Moreover, the traffic through the firewall is increased in steps of one, instead of steps of two which were seen previously.

This indicates that traffic only goes through the firewall in one direction and that something is wrong between the firewall service and the NAT service. Instead of having to perform Tcdump or debugging commands on many different interfaces, by looking at the web GUI there can be only two different causes for this error. Either the NAT service is dropping packets for some reason, or the rules in the command-and-control switch between firewall and NAT are incorrect.

![Figure 5.8: Debug NAT service down](image)

**5.3 Conclusions**

In this section, conclusions will be drawn on the goals of the analysis. It will take the analyses results from the previous section into account.

The first goal in this chapter was to explore how the system reacts in different common cases of ISPs. The complexity of the tests was increased gradually over the section. Test cases included situations with one or multiple CPEs. For multiple CPEs, the tests were divided in having them on different and on the same client switches. The results showed that, for the cases that were tested, the configuration pushed by BECS resulted in the intended flows, which were pushed to the network. Moreover, the exploration performed included a part for multi-tenancy. It was found that multi-tenancy does not result in interference between different SFCs by configuring two CPEs to have the same service.

Despite the cases explained above were successful, a few cases could not be tested, because there was no way found to test these cases with Mininet. Nonetheless, the system has been developed to deal with these situations. The first case that could not be tested was replacing or moving a CPE. In real life, network operators may have to replace CPEs, because they break. They could also move CPEs, because of new and finished contracts or because they use DHCP instead of static addresses. This could not be tested because no way was found on how to destroy a link in Mininet between switch and host and then create another link with a new host. Moreover, ISPs may provide multiple services of which some need an IP address. This means that it could be that there are multiple IP addresses on the same interface. Unfortunately, no way was found to have multiple hosts on the same port of an OpenFlow switch in Mininet and therefore this case could also not be tested.
From the tests performed it can be concluded that for the investigated situations, the system sets the correct SFC. Nevertheless, more testing and investigating is required to make sure other situations are dealt with properly as well. The reason for this is that not all situations of interest could be analysed.

The second goal of the analysis was to investigate whether the combination of BECS and the web GUI could be used as a troubleshooting tool. For this two different tests were done. The first test was a configuration error in BECS. Instead of trying to configure the network with an erroneous VNF, BECS received an error from OpenDaylight and showed this to the user directly. This way the user notices that he or she made an error quickly and can take corresponding actions to fix the configuration error immediately.

The second test tried to set a VNF that was down. The result for the end-customer is that the Internet is not reached by his devices. For a network operator normally there can be different reasons for this failure and he or she has to find out what is going wrong. This procedure often results in performing Tcpdump or debugging commands on different network elements. The results of the analysis demonstrate that from the web GUI, one can narrow down the issue significantly. For this reason, the web GUI could be used as a tool to help when troubleshooting the network.
6. Conclusions and Future Work

This chapter contains a summary and conclusions that could be drawn from the practical solution found and the analyses of the data. The chapter will also contain some future work recommendations. Because this project is the very beginning of a solution in service chaining using BECS in combination with SDN, there are some new recommendations regarding improvements and possible expansions to the software. The second section of this chapter will include this future work.

6.1 Summary and Conclusions

The purpose of the project was to create a proof of concept where SDN is integrated into the NMS BECS and where BECS is capable of performing SFC using SDN. The reason why BECS is preferred over having an SDN controller only is that BECS already knows the concept of a service. BECS can also communicate with legacy networks, which an SDN controller can not. That BECS can do so brings a general interest to the project, because it can manage hybrid networks. This property allows transitions from legacy networks to hybrid and SDN networks to be easier. There were five main problems identified which have been solved during the research and are listed below.

- How to represent SDN networks in BECS
- How can a service chain be created in BECS
- How should communication be set up between BECS and OpenDaylight
- How can the solutions found be tested without any physical elements
- How can errors in the network be found.

The problems have been approached from the bottom up, starting at how to create a test network. This way the required flows could be decided upon. Since it was not possible to use physical elements, a way of simulating the SDN network had to be found. Multiple network simulators and emulators were found, of which some were not open source. For budgetary reasons, those were not chosen. After careful consideration, it was clear that Mininet would be the best solution for this project. The first reason for choosing Mininet is that network can be scripted. Moreover, Mininet allows running any program on the network hosts that the native operating system can run. To run the VNFs, this was a necessary functionality. For other open source network simulators that were investigated, no information could be found about at least one of these properties.
Chapter 6. Conclusions and Future Work

The next problem in line that was solved was the communication between BECS and OpenDaylight. BECS already has a SOAP API, which could be used by OpenDaylight. However, OpenDaylight also has a REST API, which could also be used. The requirement for the thesis project was that there is only one-way communication between BECS and OpenDaylight. Since the information sent by BECS contains the configuration of the network and is therefore time-sensitive, the REST API of OpenDaylight was chosen to be used and extended.

When the communication between BECS and OpenDaylight was set up, the problem about how to represent an SDN network in BECS was investigated. BECS has a configuration tree that represents network elements and their corresponding interfaces below them. In [11] it was chosen to have three kinds of switches in the network, the client switch to which CPEs are connected, the command-and-control-switch to which the VNFs are connected and the gateway switch, to which the gateway(s) are connected. Because SDN networks should be represented as similar as possible to legacy networks, it was decided to add these elements to the configuration tree as well. Each switch has an external port which is connected to the uplink, which is either the next switch or a router. Clients, either VNF or CPE are represented below interfaces of the corresponding switches. Figure 6.1 shows the result of representing the left two switches in 4.2b in BECS.

![SDN element representation in BECS](image)

Figure 6.1: SDN element representation in BECS

After a representation of SDN elements in BECS had been made, a way to create a service chain had to be developed. The different services had to be identified in OpenDaylight. For this reason, a VNF object was created that is identified by its service name. By doing so, the user only has to pick an item from a list of available services in BECS to create the service chain. A wizard has been developed, which allows the user to choose a set of services from a list. These services are then
appended to a table where the user can see the order in which the services are performed. After the user has selected the services, the service names are added to an array, which internally is appended to a string. This string is sent as HTTP content to the REST API of OpenDaylight. OpenDaylight then adds the corresponding flows in the switches, to perform the SFC.

The last problem on the list was troubleshooting the system. How can be proven that the packets are going the way they are supposed to go and how can the network be troubleshooted efficiently if a malfunction has occurred? BECS shows an error message in case the user tries to set an unexisting VNF, e.g. if there is an error in the wizard or the user makes a spelling error. The BECS user is always able to change values in BECS and for this reason, it is possible that invalid values are entered even though there is a wizard for it with only valid values.

In the network, the VNFs that are used are packet bouncers. These packet bouncers also count the packets that go through them and put the counters in a database. A web GUI was developed to read the counters from the database and display them. Using the web GUI it is not only possible to see that packets take the right path through the network, but as seen in chapter 5, it is also possible to pinpoint network errors as soon as something is wrong. Normally the network operator has to log in to the network devices and use debugging commands or Tcpdump to get the information, which is cumbersome. However, using the GUI, the place where an error occurs can be narrowed down significantly. Therefore, using the web GUI could potentially save network operators time.

6.2 Future Work

As seen in section 4.3 and the previous section the set-up as in the proof of concept is not ideal. Several things can be improved, which are enumerated in this section.

Section 4.3 showed that the controller used is not ideal. It is a single point of failure, while it is also a critical entity in the network. Next to that, it is also a far more valuable target for attackers. Therefore future research should be performed on other existing controllers or a new controller combining properties of already existing ones. This way controllers can be made more resistant against attackers. Moreover, the section illustrated that there might be quite a difference between using a packet bouncer and a real VNF, which will have to be investigated. The last part of section 4.3 explained why the code should support multiple WAN connections or CPEs on the same port of a switch, and moving a host in the network. Also, it should be possible to move and replace CPEs. However, since no way was found to perform such operations in Mininet, tests on this could not be executed.

Next to issues discussed in section 4.3 about the proof of concept set-up, the following sections will describe some more future work.

6.2.1 Introduce VLAN Tags and Allow Network Layer Forwarding

Earlier some slight improvements were discussed when choosing the topology. The number of flow rules in the OpenFlow switches could be highly reduced by adding VLAN tags. Such tags would group CPEs with the same service chains together. However, what was found was that these VLAN tags get lost as soon as the packet has to go through a network layer element. A solution to this could
be adding tunnel gateways that support data link layer tunneling, to the network to prevent the VLAN tag from getting lost. There may be more solutions to this problem, which will have to be researched.

6.2.2 Locating VNFs

The way that OpenDaylight gets to know where specific VNFs can be found should be different than what is used in the test set-up. Since locating the VNFs was out of scope for the thesis project, it was assumed that BECS knows where to find them and sends that to OpenDaylight. In reality, a different entity in the network would often be used to locate and instantiate the VNFs. There are programs that can locate VNFs like OpenStack\[60\] for example. OpenStack does much more than just locating the VNFs however. It also initiates, moves, and destroys them when needed. It can keep track of any VNF in a datacenter. OpenStack could be integrated into the system developed in this thesis. By doing this VNFs can be located by either OpenDaylight or BECS and this could reduce traffic between BECS and OpenDaylight. However, the traffic that is induced right now would then be transferred to another communication link.

6.2.3 Automatically create OpenFlow switches in BECS

Currently, the only communication between BECS and OpenDaylight is from BECS to OpenDaylight and regarding network configuration. BECS is known for its zero-touch configuration, which it does with CPEs. When a DHCP message comes in for an element that BECS has an EM for, BECS automatically creates the element and pushes the desired configuration to that element. Even though a switch is not the same as a CPE, this can also be done for switches in general. In case of an OpenFlow switch OpenDaylight gets an event when a new switch enters the network, it would be possible to make it send a message to BECS and let BECS automatically create the switch. No switches were automatically created in BECS, because this would require bi-directional communication. Bi-directional communication was out of scope of the project, because this required changes in the BECS core.

6.2.4 Summary of Future Work

Much progress has been made proving that SFC can be implemented in an SDN system, and provisioned by BECS. However, there are still improvements that could be made. These improvements can reduce the amount of traffic between the BECS server and OpenDaylight server. Furthermore, adding specific entities to the network to locate VNFs could reduce the load on the BECS server, since it does not require to have knowledge about the location of the VNFs anymore. Using real VNFs could require some extra rules to in the command-and-control switch depending on the service the VNFs offer. Lastly, the amount of errors in the system could be reduced if BECS would be capable of receiving network information from OpenDaylight.
Bibliography


