SDN used for policy enforcement in a federated military network.

Erik Sørensen

Master of Telematics - Communication Networks and Networked Services (2
Submission date: June 2014
Supervisor: Øivind Kure, ITEM
Co-supervisor: Mariann Hauge, FFI
Lars Landmark, FFI

Norwegian University of Science and Technology
Department of Telematics

Title:	Evaluating Software Defined Networking for use in Federated Military Networks
Student:	Erik Sørensen

Problem description:

Software Defined Networking (SDN) is an approach to computer networking where the control plane and data plane is decoupled, in contrary to traditional networking. It is claimed to have more flexibility than legacy networks and provides abilities to innovate faster. OpenFlow is one protocol that is often used for implementation and prototyping of SDN. Federated military networks have strong focus on robustness and network utilization. These networks can be quite dynamic in nature and link capacities can vary much (from hundreds of Gbits over fiber links to a few tenths of Kbits on narrowband Satellite links). These networks must serve the needs for national traffic, local coalition traffic, and coalition transit traffic. The traffic flowing in these networks can have different characteristics and volume dependent on the types of ongoing operations. At times HD video flows from many different sensors (and thus high throughput) might be important, at other times network robustness might be of highest priority. At all times the different interests from the users (national traffic, coalition traffic and transit traffic) must be filled according to the negotiated policy.

Objective: It is advantageous to be able to change selected policies for utilization of the federated network in real time during an ongoing operation in order to meet changed requirements for network utilization. The purpose of this thesis work is to study how SDN can solve dynamic policy enforcement in a federated military network with focus on OpenFlow's capabilities and constraints.

Methodology: The candidate needs to study how policies can be enforced in a federated military network and show how SDN can be used to solve this task. Does the SDN solution show any advantages/disadvantages compared to other important techniques for policy enforcements? The candidate should also implement a proof of concept testbed utilizing SDN and Openflow for policy enforcement. To validate the approach, a minimal policy describing one or two rules should be implemented if the time allows.

Responsible professor:	Øyvind Kure, ITEM/UNIK
Supervisor:	Mariann Hauge & Lars Landmark, FFI

Abstract

This thesis looks at how Software-Defined Networking can be used to provide policy enforcement in a federated military network. SDN is a concept in computer networking where the control plane is decoupled from network forwarding devices, and placed in a centralized location. The methodology used in this work includes a literature study, a discussion and the design, implementation and validation of a test bed utilizing the OpenDaylight SDN controller. We have found that SDN can be used for policy enforcement in federated networks, and shown this through programmatically re-assigning a network tunnel to a new path in an automatic fashion using the OpenFlow protocol. Together with the implementation, we have also described through design how groups of tunnels can be moved in the same fashion, while avoiding packet loss.

Sammendrag

Denne masteroppaven er en studie som tar for seg hvordan ”Software-Defined Networking” (SDN) kan brukes for håndhevelse av definerte regelsett i federerte militære nettverk. SDN er et konsept som brukes i IP-baserte nettverk, hvor kontrollplanet er separert og frikoblet fra nettverksenheten og plassert i en sentralisert lokasjon (en server). Den vitenskapelige metodologi som benyttes i denne oppgaven er en litteraturstudie med påfølge diskusjon, samt design, implementasjon og validering av et forsøksnettverk som benytter SDN-kontrolleren ”OpenDaylight”. Vi har gjennom studien funnet at SDN kan benyttes for håndhevelse av definerte regelsett i federerte nettverk. Dette har også blitt vist ved programmatisk flytting av nettverksbaserte traffiktunneler fra en nettverkssti til en annen. Dette er muliggjort gjennom bruk av kommunikasjonsprotokollen ”OpenFlow”. Vi har også, gjennom design, beskrevet hvorledes grupper av traffiktunneler kan flyttes på samme vis, hvor en samtidig unngår unødvendig pakketap.

Preface

This study serves as the master thesis in fulfillment of the authors Master of Science degree in Telematics - Communication networks and networked services at the Norwegian University of Science and Technology.

This thesis is the original, unpublished and independent work by the author. Invaluable input and feedback have been given by supervisors Dr. M. Hauge, Dr. L. Landmark and Professor Ø. Kure during the thesis work.

Erik Sørensen
Kjeller, Norway
June, 2014

Acknowledgements

I would especially like to thank my supervisors Mariann Hauge and Lars Landmark for taking time of their busy schedules to give me feedback during the thesis work. They gave my work a critical eye, and much needed help for discussion and theoretical foundation. They also helped me to see value in the work I performed, which gave me much needed motivation when I stood alone on an island covered in bits and bytes of ODL software, sprinkled with a fine layer of python programming.

I dedicate this thesis to my parents, who have shown me 30 years of parenting done right, and my late grandmother who passed away during the thesis work. Even at old age, she always helped me with the correct spelling and use of accusative prepositions in German.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AD-SAL	Application-Driven Service Abstraction Layer
AMN	Afghan Mission Network
AO	Area of Operation
APC	Armored Personnel Carrier
API	Application Programming Interface
ARP	Address Resolution Protocol
BF	Blue Force
BGP	Border Gateway Protocol
BGP-LS	Border Gateway Protocol - Link-State
CLI	Command Line Interface
CP	Command Post
CR-LDP	Constraint Based Label Distribution Protocol
CSPF	Constrained Shortest Path First
DB	Database
DCAN	Devolved Control of ATM Networks
EGP	Exterior Gateway Protocol
ERO	Explicit Route Object
FG	Flow Group
FMN	Federated Mission Network

FRR	Fast Reroute
FSDC	Federated SDN Domain Controller
I2RS	Interface to the Routing System
IETF	Internet Engineering Task Force
IGRP	Interior Gateway Routing Protocol
IP	Internet Protocol
IRC	Internet Relay Chat
IRTF	Internet Research Task Force
ISAF	International Security Force in Afghanistan
JVM	Java Virtual Machine
LAN	Local Area Network
LLDP	Link Layer Discovery Protocol
LSP	Label Switched Path
MD-SAL	Model-Driven Service Abstraction Layer
MN	Mission Network
MPLS	Multi Protocol Label Switching
MPLS-TE	Multi Protocol Label Switching - Traffic Engineering
NATO	North Atlantic Treaty Organization
NCS	Network Control Server
NEC	Network Enabled Capability
NHOP	Next-Hop bypass tunnel
NNHOP	Next-Next-Hop bypass tunnel
NOS	Network Operating System
ODL	OpenDaylight
OF	OpenFlow

OFC	OpenFlow Controller
ONF	Open Networking Foundation
OSGi	Open Service Gateway initiative
OSI	Open Systems Interconnection
OSPF	Open Shortest Path First
OVS	Open vSwitch
OVSDB	Open vSwitch Database Management Protocol
PBB	Provider Backbone Bridge
PCN	Protected Core Networking
PCS	Protected Core Segment
QoS	Quality of Service
REST	Representational State Transfer
RIP	Routing Information Protocol
RSVP	Resource Reservation Protocol
RSVP-TE	Resource Reservation Protocol - Traffic Engineering
SAL	Service Abstraction Layer
SDN	Software-Defined Networking
SLA	Service Level Agreement
SLO	Service Level Objective
SPF	Shortest Path First
STO	Science and Technology Organization
STP	Spanning Tree Protocol
T	Tunnel
TACOMS	Tactical Communications
TCP	Transmission Control Protocol

TE	Traffic Engineering
TG	Tunnel Group
TTL	Time-to-Live
UI	User Interface
URN	Uniform Resource Name
VLAN	Virtual Local Area Network
VM	Virtual Machine
WAN	Wide Area Network
XML	Extensible Markup Language

Chapter 1 Introduction

Software-Defined Networking (SDN) is a network technology that is starting to gain traction in the network industry, after several years of research in academia [1]. SDN is a concept where the control and forwarding planes of network devices (e.g. switches and routers) are separated, and control logic is put in a centralized location. A specialized communication protocol is used between the control and forwarding plane, where OpenFlow (OF) is the most widely implemented.

Federated military networks are federated networks used in a military setting where several nations interconnect their networks for the purpose of sharing information and network resources.

1.1 Motivation

In federated military networks, such as e.g., Federated Mission Networks (FMNs) [2] and Protected Core Networking (PCN) [3], there is a need to share network resources between coalition partners. One of the main motivations behind the development of federated military networks is that the different nations shouldn't have to bring all the networking equipment themselves. Instead they can share capacity among each other. In traditional Internet Protocol (IP) networks, the majority of bandwidth resources are idle and unused. Around 40% utilization of available network resources is a common value [4], but could often be much less. For instance in Uninett's[1] backbone network, where many of the links are well below 5% utilization[2] [5]. We see that a greater utilization of shared network resources in the battle space could simplify the deployment of network resources. It will also lower the cost for all nations involved, if these resources could be used more effectively. There is no secret that the establishment of a military infrastructure based network in a conflict area is both difficult and expensive. The less links and nodes that have to be established, the better it will be for all coalition partners.

[1]The Internet service provider for the Norwegian universities and research establishments.

[2]Network fluctuations will of course impact the network immensely, so higher values could be found during peak hours.

Policies can be defined as a set of rules that govern how traffic should be treated in a network. Policies can cover such areas as access control and security, dependability and utilization. The goal of obtaining better network resource utilization is not a new idea. This can be solved today through the use of a mechanism called Traffic Engineering (TE), which has been used in a number of years. TE helps to provide multipath forwarding. When considering TE and policies, one could say that policies are the rules that govern how TE, as a mechanism, should operate.

One of the best known and widely deployed techniques for TE is the use of Multi Protocol Label Switching - Traffic Engineering (MPLS-TE). This is the preferred technique for core and Internet service providers in the pursuit of better network utilization. However, it has been proposed that MPLS-TE services, can be better provided through the use of SDN. Google has one of the largest and most well known implementations of SDN in their B4 internal datacenter WAN [4], for the purpose of dynamic traffic engineering and better link utilization.

While Googles usage scenario is quite specific and exclusive to their service needs, we propose that SDN could be used in federated military networks with better results and flexibility when it comes to policy enforcement and network utilization, than what current technologies can provide. This thesis will therefore look into the use of SDN as a tool for policy management and enforcement of policies in federated military networks.

1.2 Derived Problem Description

Based on the problem description (ref. title page) the derived problem description of this thesis is: *How can SDN be used to provide policy enforcement in a federated military network? An implementation of a simple policy rule should be created to show how policy enforcement can be solved in SDN.*

1.3 Thesis structure

The thesis structure is as follows:

- Chapter 1 is an introduction to, and motivation for, this thesis.
- Chapter 2 will present relevant theory on the subject. This chapter will provide a backdrop and reference to concepts discussed later.
- Chapter 3 will provide a discussion on the use of SDN in federated military networks, and relevant policies that can be implemented in such a network.

- Chapter 4 will continue on the work from chapter 3 and present some possible designs for policy implementations in a federated network, specifying closer how it can be solved through the use of SDN.
- Chapter 5 will present the work done on the implementation of a test bed, together with testing and validation.
- Chapter 6 will present experiences gathered through the implementation work.
- Chapter 7 will present the conclusions from the thesis work.

Chapter 2 Theory

This chapter will provide a short introduction to theory which is considered important for the understanding of the work presented in this thesis. First an introduction to SDN will be presented. This is to give an understanding about how SDN differs from legacy networking techniques[1]. This will be followed by an introduction to the OF protocol. The chapter will then give a description of the MPLS-TE protocol, which will be used for comparison between SDN and legacy networks. An introduction to policies in networking will also be given, together with an introduction to federated military networks.

2.1 SDN

SDN is a concept in networking where the control and forwarding planes are decoupled, as opposed to traditional networking where the control plane is located on the physcial switch or router (ref figure 2.1). In traditional networking, a multitude of (distributed) protocols run on switches and routers. These are all implemented in a local control plane on embedded hardware. This means that you would also have to change out the network device (hardware) if you need new functionality.

[1]The reader is assumed to have a good understanding of traditional networking techniques and technologies.

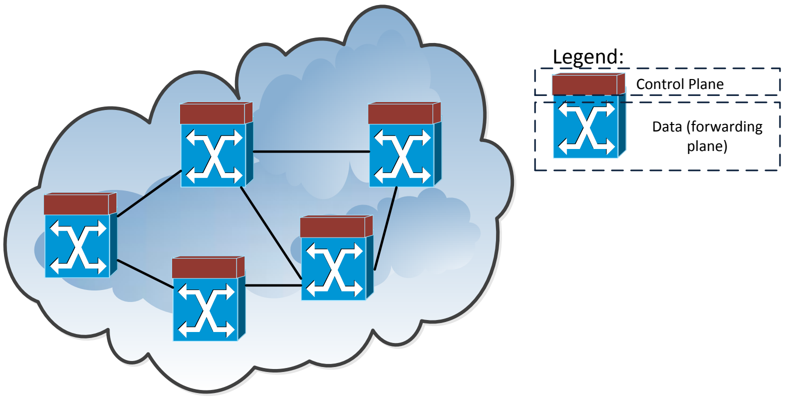


Figure 2.1: Traditional Networking

In SDN, the control plane is moved to a centralized software based controller (ref. figure 2.2). The controller can run on an enterprise grade server, providing more processing power than embedded hardware solutions. Control logic can also be updated through software, which makes it easier to update the network if new functionality is needed.

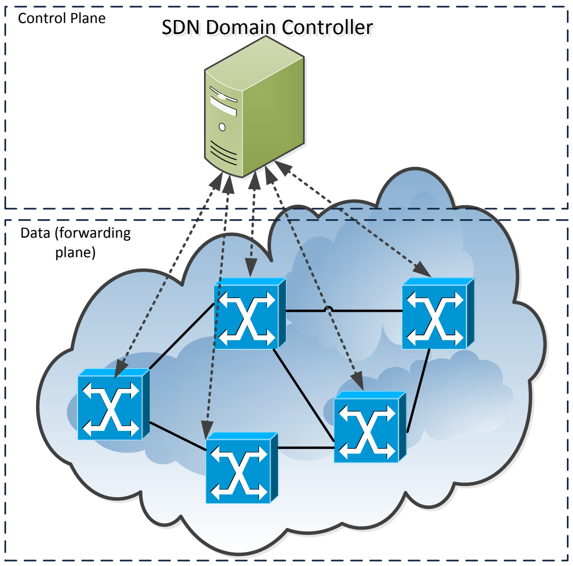


Figure 2.2: SDN Networking

The controller in SDN instructs the networking elements by updating forwarding tables, and thereby controlling how the traffic flows. This is done through an open interface (e.g. OF[6] or NetConf[7]). Messages containing changes in the topology will also be sent over this interface. There are no standardized solutions for topology discovery, but existing protocols that provide such services can be implemented in the controller. E.g. Link Layer Discovery Protocol (LLDP) [8], Border Gateway Protocol - Link-State (BGP-LS) [9] or Interface to the Routing System (I2RS) [10]. Forwarding rules can be inserted either in a proactive, or reactive manner. Proactive means that the flow tables are populated with flow rules before new packets arrive. Reactive means that newly arrived packets not matching existing forwarding rules are sent to the controller for processing. The controller process the packet and installs new forwarding rules on the switch/router. The network devices could therefore be viewed as dumb devices, that only does table look ups to see what action(s) it should perform on the packet[2]. As forwarding of packets are based on quick look ups in forwarding tables, traffic will be forwarded at line speed as long as flow rules are installed.

SDN in it's current form is a concept coined mainly by researchers at Stanford and Berkley University. There are however several earlier research efforts that have led up to today's SDN concept. The most prominent examples are: *Open Signaling* [11], *Active Networking* [12, 13], *Devolved Control of ATM Networks (DCAN)* [14], *Clean Slate 4D Project*[15], and more recently *Ethane* [16]. A thorough walkthrough of these efforts can be found in [1].

SDN can in its simplest form be viewed as having three main layers (or planes). A generic architecture for SDN can be seen in figure 2.3. The forwarding/data plane is populated with network devices that does the actual forwarding. Above this plane is the control plane. This is where the SDN controller resides. The SDN controller communicates bidirectionally over a so called southbound interface to the network devices. The most common interface used today is the OF protocol. The controller can also be viewed as layered system with a Network Operating System (NOS) at the bottom layer. The NOS has a global view of the network and communicates with the network devices. The network virtualization layer above provides an abstract view of the topology, and a control application above serves a specific task or service (e.g. an LLDP implementation.) There could also be several controllers in the control plane that operates together in a fail over or load sharing scenario.

The top layer in the SDN architecture is called the application plane. This is where business specific applications resides. They can be used to provide network services by communicating to the controller over an northbound Application Programming Interface (API). There is very little, or none, standardization of northbound APIs. It should therefore be viewed as controller specific. A typical example of an SDN application embedded in this

[2]OpenFlow enabled switches could also operate in a hybrid mode, where a local control plane assumes control if communication to the controller is lost.

plane could for instance be a Border Gateway Protocol (BGP)[17] routing deamon. I.e. an application that mimic the behavior of the BGP routing protocol.

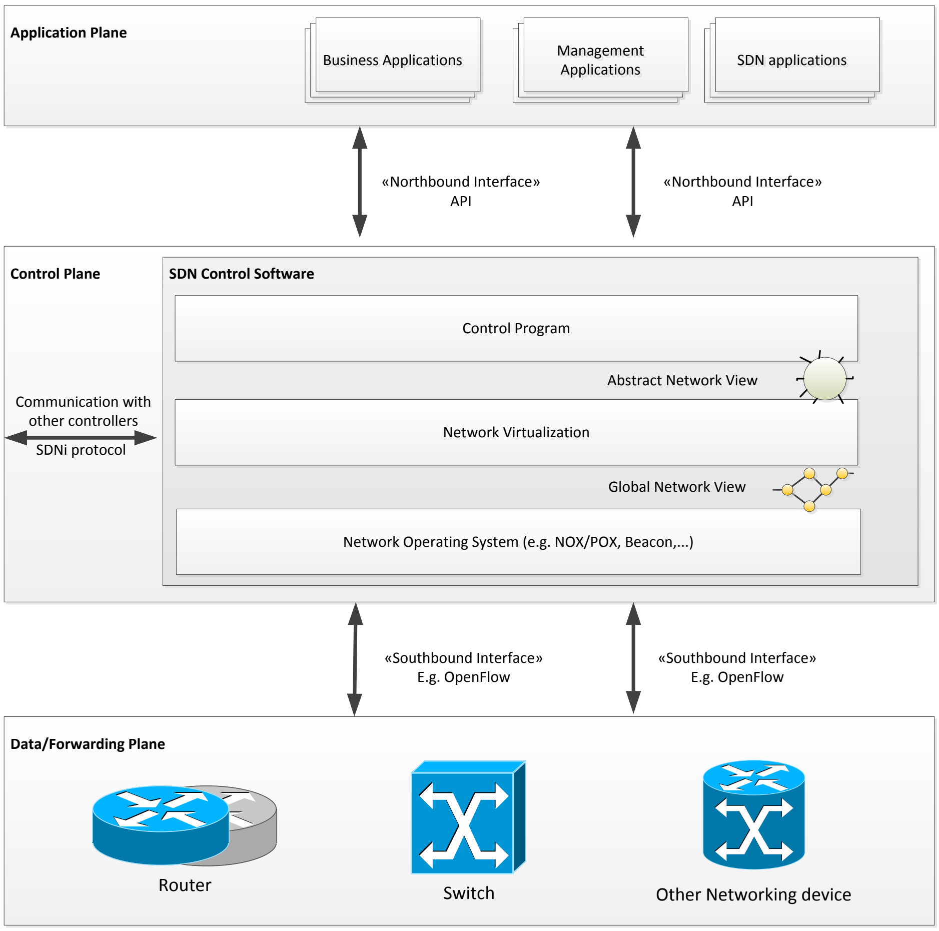


Figure 2.3: Software-Defined Networking Architecture. Figure based on [18, 19]

2.1.1 OpenFlow

The most common and widely deployed interface towards networking devices is known as the OF protocol. This protocol were originally developed at Stanford university [6], and has become a de facto industry standard. In later years, the protocol has come under control and development by the Open Networking Foundation (ONF). Several industry leaders are active in the development of the protocol, and have started producing OF compatible network devices.

OF enabled switches have a number of flow tables that are connected in a pipelined fashion. Incoming packets are sent through the pipeline until the header values matches a flow rule's *match fields*. Match fields can be from layer 1 to layer 4 (ref. the Open Systems Interconnection (OSI) model), and can for instance be: Input port, Ethernet address, IP address, Transmission Control Protocol (TCP) port or a tag value (e.g. Multi Protocol Label Switching (MPLS) tag). It can also be a wild card value. When a match is found in one of the flow tables, a corresponding action is performed. E.g. forward, push/pop tags, drop packet etc.

There are also two other tables in an OF enabled switch. A meter and a group table. Meter tables are used to provide Quality of Service (QoS) functionality such as rate-limiting. Entries in a meter table are based on three components: A *meter identifier* that identifies one unique meter, a *meter band* that specifies the rate of the band and how to process the packets, and *counters* that are incremented each time a meter band is used [20, § 5.7.1].

Group tables are used to perform actions on a group of flows. For instance to forward on the first live port in a group of ports (fail over), or to forward on all ports in a group of ports (multicast).

OF uses a number of *match fields* and *action types*. These are defined in the OF specification, where the latest ratified version is 1.3.3 [20]. Table 2.1 and 2.2 lists the required match fields and action types for the device to be in compliance with the specification:

Match Field			
Layer 1	**Layer 2**	**Layer 3**	**Layer 4**
Input port	Ethernet src address	IPv4/IPv6 protocol number	TCP src port
	Ethernet dest address	IPv4 src address	TCP dest port
	Ethernet type	IPv4 dest address	UDP src port
		IPv6 src address	UDP dest port
		IPv6 dest address	

Table 2.1: Required OpenFlow Match Fields, from [20]

Action	**Description**
Output	Forwards packet to specified port
Drop	Not explicitly specified, but packets without an output action should be dropped
Group	Process packet through the specified group in the group table

Table 2.2: Required OpenFlow Action Fields, from [20]

There are also optional action types defined in the OF standard. Most notably are the *Set-Queue* action used for QoS operations, and *Push-Tag/Pop-Tag* actions that can be used on Virtual Local Area Network (VLAN), MPLS and Provider Backbone Bridge (PBB) headers. The Time-to-Live (TTL) values for IP and MPLS can also be changed through use of a *Change-TTL* action type.

2.1.2 Northbound APIs

Northbound APIs are the connection between network applications and the SDN controller. The applications could for instance be policies engines, business applications or applications concerned with network control. The APIs are meant as a way for the applications to communicate with the controller. To acquire information from the controller, or for giving the controller instructions for network operation.

At the current time there are no standardized northbound APIs, as compared to OF or other southbound protocols. Northbound APIs from the SDN controllers are mostly Representational State Transfer (REST) based, such as in the OpenDaylight (ODL)[21] controller. The ONF has however set down a working group for standardizing the north bound interfaces [22]. If this work comes into fruition, and is agreed upon by the developers of SDN controllers, SDN applications could operate towards any SDN controller. Much in the same way as SDN controllers can operate any network devices that supports the OF protocol (or any other south bound protocol for that matter, e.g. NETConf, Cisco onePK[23], or Open vSwitch Database Management Protocol (OVSDB)[24]).

2.1.3 Slicing in SDN

An important concept in SDN is *slicing*. Slicing is in its most basic form descriptive of how SDN can be used to share network resources between different users. I.e. *"You will get a slice of the physical network to use for your traffic"*. However, the use of the term has shown to mean different things. One form of slicing is to give specific users of an SDN controller the possibility to control flows for just a few ports or nodes (i.e. *"You can use only these resources"*) [6]. Another view is to provision a user with the possibility to only create flows with a specified set of flow rule match fields [16]. I.e. *"You can only create flow rules which matches addresses in this subnet"*. A third and more specific use of the term comes from the OF specification [20] where slicing is used to describe bandwidth sharing between different queues on the same output port.

Slicing in SDN networks has received serious traction, especially in data centers where it is used to create "multi tenancy" networks. This is a concept where a tenant (a data center customer) is allocated a virtual network on the physical infrastructure of the data center. This allows the tenant full control of the resources he has been granted without interfering with other tenants of the data center. The tenant will only see "his own network" between the compute and storage nodes, even though it runs on physical infrastructure shared between several tenants.

2.1.4 Google's SDN Use Case

At the current time, Google has the most well known and largest implementation of SDN in a live production network. For the motivation of this thesis, we see it as important to look into why Google chose an SDN implementation, and how they implemented it.

Google has an internal Wide Area Network (WAN) that connects all of their data centers across the globe. This was made fully SDN capable in 2011-2012, and is called *B4* [4]. Googles motivation for this was to: Increase utilization of the expensive WAN links, instead of over-provisioning bandwidth resources two-fold or more. Edge rate control for competing applications and provide dynamic bandwidth reallocation in case of failures or shift in demands.

Their data centers have a substantial bandwidth requirement, and has more traffic than their public facing WAN. There are some sides to this network that makes it unique compared to other large WANs. First off, Google has complete control of everything on the network. This includes applications, servers and Local Area Networks (LANs). Secondly, their data centers perform large-scale data copies between data centers. Google's SDN architecture includes one SDN controller for each data center, and a centralized TE controller for the whole WAN. This solutions has shown that they can reach ~95% network utilization.

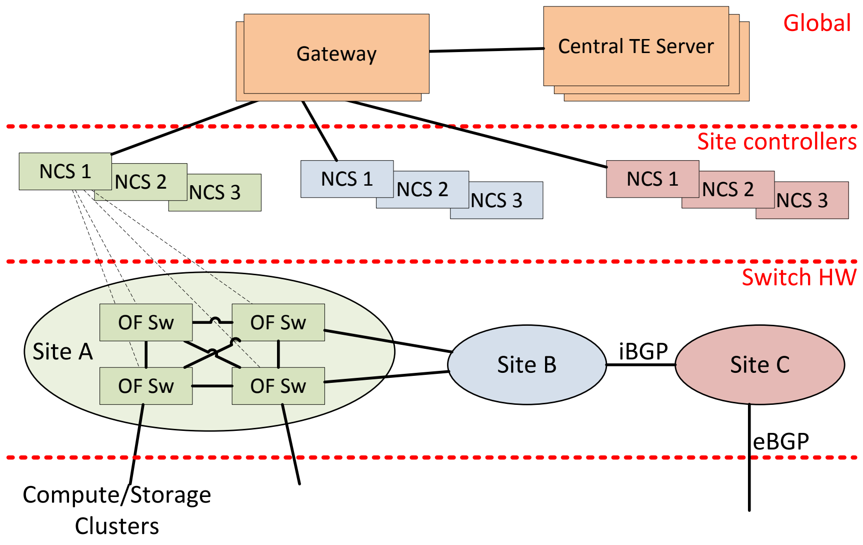


Figure 2.4: B4 Architecture, from [4]

The B4 WAN architecture (ref. figure 2.4) implements TE as a routing overlay between sites, with BGP routing between each cluster (BGP routers not shown in figure). Google chose this separation as a fail safe so that in case of a failure in the SDN controllers or TE server, the network could go back to Shortest Path First (SPF) routing. OpenFlow Controllers (OFCs) and OF switches communicates over a separate out-of-band network. A gateway is used between sites and the TE server to consolidate topology changes. The abstraction created through this simplifies the graph used by the TE algorithm.

TE provides edge rate limiting, multipath forwarding and dynamic reallocation of bandwidth in case of failures or shifting resource demands. In operation, the TE server will create Flow Groups (FGs) that matches a *source site, dest site, QoS* tuple, Tunnels (Ts) as a sequence of sites using IP-in-IP encapsulation and Tunnel Groups (TGs) which map FGs to a set of tunnels with a weight value to show the fraction of traffic from the FG to be forwarded over each tunnel.

To provide dependability in the network Google employs a number of strategies: Software failures are seen as the largest cause of failures, so moving control from the data plane to the control plane will mitigate this. Network Control Servers (NCSs) are replicas running on different servers with leader election for primary controller using Paxos [25].

2.2 Inter-Domain State Distribution in SDN

Forwarding traffic between different network domains (or segments) of a larger network is an important area in networking. In the Internet, this is mainly done using BGP as the standard Exterior Gateway Protocol (EGP). SDN is mainly an inter domain network technology, and would not work *straight out of the box* in a federated network with autonomous network segments, each with it's own controller.

One challenge with SDN is the problem with inter-domain state distribution. Where state information is shared between interconnected domains. SDN controllers in its current form does not have any knowledge of the network out side of its own domain, unless this information is given. This challenge has seen some research. An obvious example concerning this is the work done by the Internet Research Task Force (IRTF) on proposing the SDNi protocol. This protocol is meant to provide a standardized information exchange between controllers from different domains. At the time of this writing, we have understood that the protocol is still only in draft status, but work continues in this area.

The question of federated SDN networks and the work on inter-domain state distribution were also raised during the OpenDaylight[3] summit 2014 during a panel discussion[4] with the lead developers of the OpenDaylight project (see section 5.1.1). They acknowledged this as an important area that needs further research, but that it at the current time is not yet implemented. This was proposed as a project that could be done in the frame of the OpenDaylight project, but that it would most likely (in the near time) be a question of getting different OpenDaylight SDN controllers to talk together and share state-information, before it's likely to see SDN controllers in general communicate.

Based on this it seems clear that SDN, at the current time, will have challenges if it's to be used for inter-domain routing. However, BGP can be implemented in SDN as an application and be used to interconnect different SDN domains in the same manner as BGP would interconnect traditional network domains. But it will not alleviate the inherent problems we see with BGP today (i.e. large BGP routing tables).

2.3 MPLS-TE

When discussing the use of SDN as a solution for policy enforcement it's important to compare it with how the same issues are solved in legacy networking. One very important technology to provide network traffic steering and link dependability is the MPLS-TE protocol. The following is an introduction to MPLS-TE were key attributes are used later for comparison against SDN.

[3]OpenDaylight is a project between several network industry giants and the Linux Foundation to create an open source SDN controller that can be used in SDN development and implementations.

[4]The panel discussion can be viewed here: https://www.youtube.com/watch?v=mwCe7bxvcvE

MPLS-TE is an extension to the MPLS [26] protocol that adds TE to an MPLS network [27]. TE is a technique for steering and controlling the flow of traffic in a network. The main purpose of TE is to get the best possible utilization of available network resources. In normal plain-vanilla IP networks this is not possible due to the fact that different flows will not be routed over different paths between the same source and destination nodes. MPLS-TE enables this through multi path routing. MPLS-TE is the dominant technology in the industry for TE. This is especially true after the move from ATM and Frame Relay to pure IP networks, as these legacy link layer technologies had the possibility for multi path forwarding.

In normal IP networks, the traffic is routed on the basis of getting it from sender to receiver as fast as possible. IP networks are therefore said to have a *least-cost* forwarding paradigm [27]. This is done through assigning a cost value for all links throughout the network. This cost can be a single metric value assigned to the link (as used in Open Shortest Path First (OSPF) [28, 29]), a composite metric (as used in Interior Gateway Routing Protocol (IGRP)) or the hop count (as used in Routing Information Protocol (RIP) [30]). This cost value does however not take bandwidth into consideration, but merely *distance*[5]. The effect of this will be that a router could continue to forward traffic onto a link that already drops packets. As this is considered the shortest path to the destination.

Consider figure 2.5, where a pure IP network is running the OSPF routing protocol. All links in the network have a 10Mbps capacity. If traffic should flow from R3 to R7, R3 will set up a path [R3-R4-R7], as this is the shortest path. However, a lot of traffic flows between R3 and R7, and at some point the nodes will start dropping packets. At the same time, nothing is routed over the path [R3-R5-R6-R7]. This creates a huge under utilization of link capacity in the network.

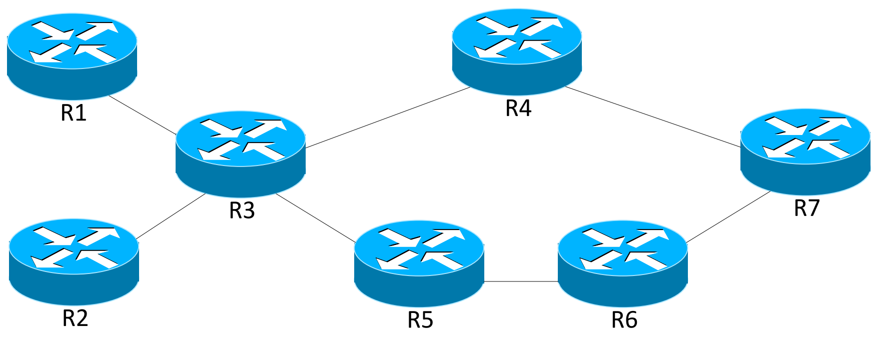


Figure 2.5: Forwarding Network

[5]There has been research into QoS routing to mitigate this problem, but nothing has come into widespread use on legacy network equipment. An example is *Pathlet Routing*[31].

MPLS-TE provides a solution to this. With MPLS-TE you can map specific traffic to a MPLS route. In MPLS-TE, the ingress router of a traffic engineered path through the network is called a *head end router*, the egress router of the same path is call a *tail end router*. The head end router will calculate the most efficient path from head to tail based on link attributes (delay, jitter, etc.) and available bandwidth. It will the create an Label Switched Path (LSP) path between the head and tail following the calculated optimal route. Routers that the traffic flow traverses will push/pop MPLS tags (as in normal MPLS networking), and label switch the packets forward. Bandwidth for this particular LSP will be reserved on links between adjacent routers as decided by the head end router. In this way (following the above mentioned example), router R3 can load balance traffic between itself and R7 on both path [R3-R4-R7] and [R3-R5-R6-R7]. Creating tunnels on both physical paths.

The head end router employs a path calculation algorithm to calculate MPLS-TE tunnels. To be able to do this calculation, it needs to have some understanding of what the topology of the network looks like. To build a topology view, a link state routing protocol have to run on the network. Each router on the network will flood their understanding of the topology to all others. The constraint based resource information that the link state routing protocol carries in the MPLS-TE network are:

- *TE Metric:* A metric different than (for instance) the OSPF cost which can be used to create a topology that differs from the IP topology.
- *Maximum Bandwidth:* The total bandwidth of a link.
- *Maximum Reserved Bandwidth:* The maximum amount of bandwidth that can be reserved on a link.
- *Unreserved Bandwidth:* The total bandwith of a link that's not reserved for other tunnels.
- *Administrative group:* A group that can be customized for use by the network operator. Could have any meaning that the operator of the network chooses.

When a new tunnel is calculated and ready to set up, a Resource Reservation Protocol (RSVP) Path message will be sent down the path towards the tail end router. A RSVP Resv message will be returned. This ensures that the topology hasn't changed during the path calculation, and that the bandwidth that has to be reserved to the new path is still available. The RSVP Path message finds its way through the network using the Explicit Route Object (ERO) from MPLS. To use RSVP in a MPLS-TE scenario the RSVP protocol has been extended to support TE, and is now known as Resource Reservation Protocol - Traffic Engineering (RSVP-TE). Other signaling protocols have also been proposed, such as Constraint Based Label Distribution Protocol (CR-LDP), but the Internet Engineering Task Force (IETF) have decided to focus its efforts on the RSVP-TE protocol [32].

2.3.1 MPLS-TE Protection Schemes

MPLS-TE has two main protection schemes built into the protocol standard. Fast Reroute (FRR) and path protection.

FRR

FRR is a local protection scheme, and comes in two flavors. Link and node protection. In link protection, a backup tunnel is created beforehand in the case a link on the path should fail. This backup tunnel is called a Next-Hop bypass tunnel (NHOP). With link protection only the link is protected, so the NHOP will create an alternative path to the next-hop node. With node protection, a Next-Next-Hop bypass tunnel (NNHOP) is created to provide an alternative path to the next-next-hop node. This means that if the next-hop node should fail, the traffic on the path will be routed around this node. An illustrative example of the two methods can be seen in figure 2.6 and 2.7 respectively.

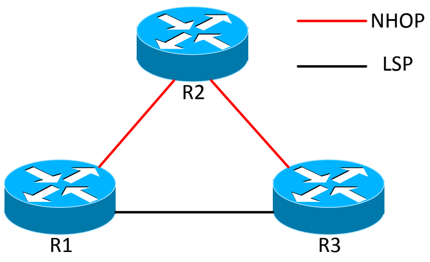


Figure 2.6: Link Protection in MPLS-TE, from [27]

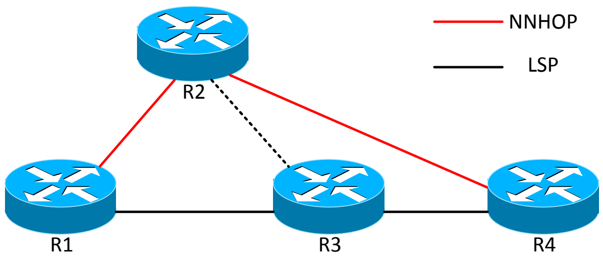


Figure 2.7: Node Protection in MPLS-TE, from [27]

Path Protection

Path protection is different from the FRR scheme, as it protects a full path end-to-end. The path protection scheme creates a separate back-up path end-to-end, which is not influenced by failures on the primary LSP [33]. Switch over to the back-up LSP, from the protected LSP, is done at the head-end router immediately after signaling of a broken link or node is received. Several signal functions can be used for this.

In some implementations of MPLS-TE, it's also possible to create several back-up LSPs. This can for instance be done in Cisco IOS Release 12.3(33)SRE and later versions, which provide up to eight back-up LSPs.

2.4 Introduction to Policies

When looking at policy enforcement in a federated network, it's important to have an understanding of what *policies* actually means in this context, and to have a common semantically understanding of terms used.

Policies are used in the management of networks. In its shortest form, you could say that the purpose of a policy is to create predefined rules that specify actions in response to a defined criteria [34]. The IETF has created an internet-draft on how a *Policy Framework Architecture* should look like. This provides an overview of the use of policies in a high-level and generic way that seems applicable to build understanding of the topic.

To be able to define and use policies, the IETF defines some components of a system as critical [34]:

- The ability to define and update policy rules
- The ability to store and retrieve rules
- The ability to decipher the conditional criteria of a rule
- The ability to take the specified actions of a rule, if the conditional criteria are met.

The actual policy rules are at the center of a policy based architecture. In the management of a network you should be able to create rules on the basis of stated agreements and objectives of the functionality of the network. Such as Service Level Agreements (SLAs) and Service Level Objectives (SLOs). These have to be translated into specific rules that govern the policies. A simple policy rule could for instance be:

- *Premium traffic between A and B shall receive a minimum bandwidth of 10 Mbps.*

This is a QoS based rule, but is not specific enough so that it can be directly used in a network. If we translate this rule into the context of an OF based SDN network, a lot of rules and instructions would have to be set. Defining ”premium traffic between A and B” could be done through the use of a flow rule that matches A and B with a IP source/destination pair, with a specific TCP port number as ”premium” traffic. The notion of a minimum bandwidth of the flow could be solved through attaching other flows to meter table entries that rate limit the traffic up to a certain amount, so that 10Mbps is always available to the flow between A and B.

In legacy network management there has been severe problems with implementing good high level policies onto low level vendor specific equipment. Changes in policies has often meant that many network devices would have to be reconfigured, either through manual reconfiguration done on a Command Line Interface (CLI) or through scripting. Many network managers have trough time and testing developed their own scripts to manage their specific network. One could say that the management is based on a high level of experience with managing the network. If the behavior of the network changes, an experienced network administrator can in many cases ”see” where the problem lies intuitively. This is however not a good solution, as it will depend highly on specific individuals operating the network over a long period of time.

There has been work on policy languages for SDN that could be used to translate high level semantics into low level network device instruction. Examples of these are *Procera*[35, 36] and *Frenetic*[37], which are essentially network programming languages. This work will however not be regarded further in this thesis as it is not directly relevant, but rather mentioned to provide an understanding of related work in this research area.

2.5 Federated Military Networks

This section will provide some context regarding the use of the term *federated network*, with emphasis on *federated military networks.*

Federated networks as such are not specific to the military, but could be used to describe any network where several network segments from different operators are interconnected to create one common service or resource. Just interconnecting two networks will not make them federated as such[6].

A federated military network is a network built from the interconnection of several networks from different nations. The purpose of these networks are two fold: Firstly, they serve information sharing between the different nations in a coalition. Secondly, they provide an unified mesh so that the different nations can share network resources when

[6]In the rest of this thesis (unless specified otherwise), a the term *federated network* is used to describe a federated military network

needed. This means that the network should be able to transport flows for three main classes of traffic, as defined here:

- *National traffic:* Traffic that flow in the network of a specific nation, and only between nodes of that nation.
- *Local coalition traffic:* Traffic between nodes of different nations in the same coalition. This traffic traverses the network of two or more nations, but only in the same theater of operation.
- *Coalition transit traffic:* Traffic that traverse through a different nation's network towards a remote destination. The remote destination is considered to by outside the federated network.

Federated networks (or military networks in general) are usually built up by a multitude of different network bearers. The access network close to end-hosts are usually built up from high capacity fiber links. Core elements between nations and access networks could also be interconnected through fiber, but is often based on radio links (radio relays and SATCOM). Lower capacity radio links are also used between access networks and remote end-hosts, such as forward operating bases and individual troops.

To exemplify federated military networks we introduce two examples from different North Atlantic Treaty Organization (NATO) bodies. *Mission Network (MN)* and *PCN.*

2.5.1 Mission Networks

MNs are a breed of military networks that are meant to be deployed on a per mission basis. The main purpose of these networks are to interconnect different coalition partners working together in the same Area of Operation (AO), which are also connected back to their ”home networks”, ref. figure 2.8. There has in later years been a turn, were the expression ”Need to share” slowly takes precedence over ”Need to know”. The latter expression, being the mantra of military information security for many years. This is because one has seen the need to faster and more reliably share information in a multinational AO. E.g. in a battlefield management system, for the purpose of avoiding blue-on-blue scenarios[7].

These networks main focus has been to create interoperability between coalition partners. An example of this is the Afghan Mission Network (AMN) created between coalition partners in the International Security Force in Afghanistan (ISAF). At the core of this network is an ISAF secret security level. This is a mission specific security level on the same level as NATO secret. Directly connected to this are national networks, also at an ISAF secret security level, ref. figure 2.8. Through the use of information exchange gateways,

[7]I.e. friendly fire

these national networks are connected to their own national home networks at a standard NATO secret level. The same is seen with the NATO Mission Security Domain, which connects to NATO's own backbone network at a NATO secret level. The information exchange gateways should be viewed as middle-boxes, which only allow some information to flow in either direction.

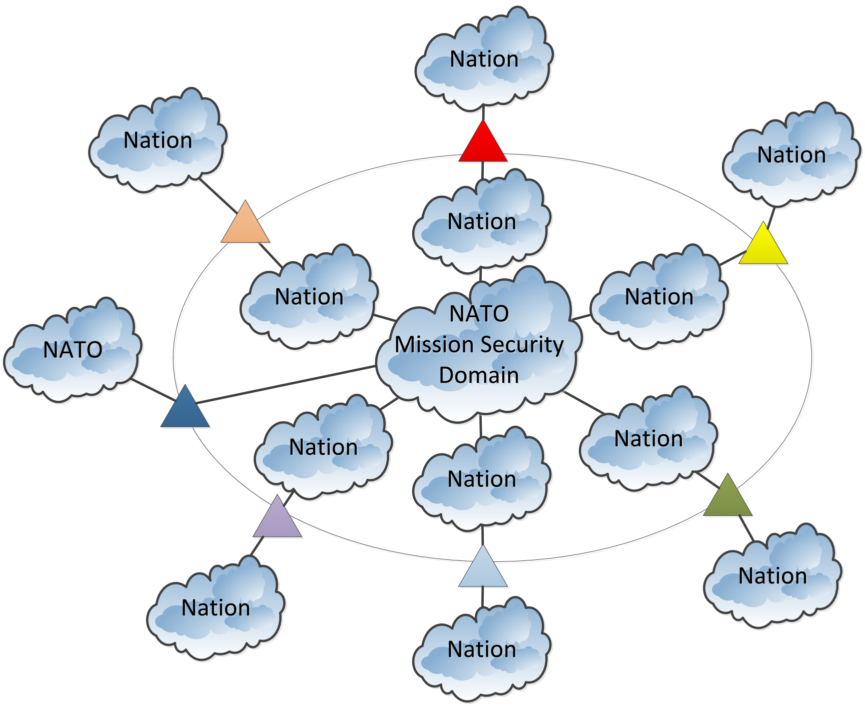


Figure 2.8: AMN Interconnections. Triangles on links are information exchange gateways, from [38]

Building on this, NATO has been working on a standardization called FMN[8]. The purpose of this is to create a standard that all coalition partners can build on, from mission to mission, instead of building a network each time from the ground up. FMN should be considered as a framework for later mission networks [2].

[8]Formerly known as Future Mission Network.

> “The aim of the concept is to provide overarching guidance for establishing a federated Mission Network (MN) capability that enables effective information sharing among NATO, NATO nations, and/or non-NATO entities participating in operations.”
>
> *Quote from [2]*

What you would normally see in such networks is that the different nations bring their own networking equipment to an AO, and interconnect through a common backbone. The backbone is usually, or should be, created in part by the first nation on the ground so that newly arriving nations quickly can connect to this. The different nations are some times co-located, especially in the beginning of a campaign, so fiber or other high bandwidth links can be utilized. Later, many of the connections from site to site will normally go over satellite as bases are set up further into the battle space.

2.5.2 Protected Core Networking

PCN is an example of a federated military network. It is a concept under research in the Science and Technology Organization (STO) in NATO, which is intended to ”implement a flexible transport infrastructure that supports military operations based on a Network Enabled Capability (NEC)” [3]. The purpose of this is to enable coalition partners to interconnect and share infrastructure without losing control over their own network[39]. Individual nations should still be able to prioritize their own traffic, but at the same time be able to share idle capacity with partnering nations. The backdrop for this concept, is an increased need for bandwidth in multinational military operations. Following this, there has been observed that the full infrastructure capacity in the networks (all networks from the different nations as a whole) is not utilized all the time. Therefore, it is a possibility to share this infrastructure between the different nations, should there be increased needs for capacity from a given entity. This should be preplanned as a part of the operation planning for a mission, to make sure that the coalition as a whole has enough network resources. This way, some nations could enter a theater with very little transport network equipment of their own. It could in this sense be viewed as an effort towards, or as an example of, a MN[9].

[9] There are infact ongoing initiatives in NATO to allign the efforts in FMN, Tactical Communications (TACOMS) and PCN as these projects are related and in some way directed towards a common goal.

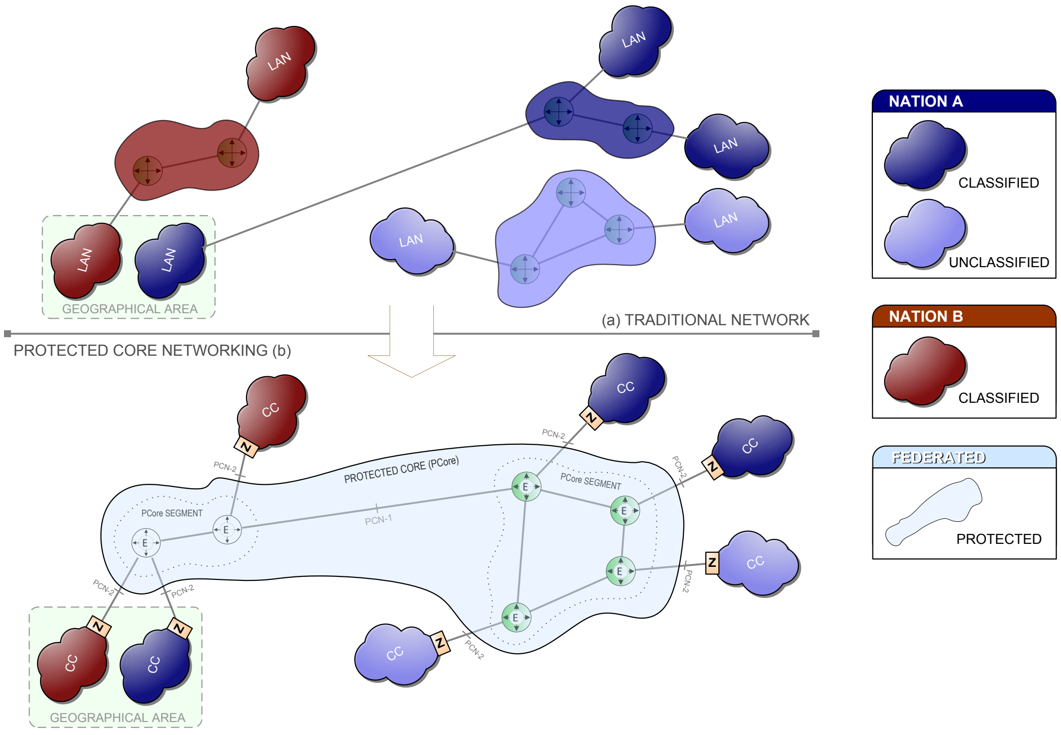


Figure 2.9: Comparison of PCN versus Traditional Military Network, taken from [3]

PCN defines interfaces between what is known as Protected Core Segments (PCSs), and between Colored Clouds and PCS (see figure 2.9 for details). PCS is a segment of a ”black side” network, and could be regarded as one part of a bigger multinational network operated by one single nation or entity. The colored clouds are ”red side” networks. A way to look at this concept is that encryption is moved as far out in the network as possible, and everything on the ”black side”, the PCSs, are considered as interconnected infrastructure segments (domains). This placement of encryption mechanisms (as far out as possible) helps in the way that all transport network capacity can be shared between networks of different security levels.

The work on PCN has also proposed some guiding principles, as stated in [39] and listed in table 2.3:

Guiding Principle	**Explanation**
Differentiated services	Similar to concepts in DiffServ [40, 41] in the Internet architecture. Different flows should get different service through the network based on marking.
Superior Management and Control	Support for high availability and security, globally in the network.
Superior knowledge	A need for a real-time view of the network. This includes knowledge of all metrics concerning the network, from traffic to security.
Superior protection	E.g. to remove unauthorized traffic immediately, and to give proper awareness to network operators.
Support of Dynamic and Federated Environments	Keeping the network interoperable between coalition partners. Should support seamless relocation of users (nodes) and network elements.

Table 2.3: Guiding Principles in PCN

Chapter 3 Discussion

The problem that this thesis tries to tackle, as briefly described in the introduction (ref. chapter 1), is to look at how SDN can be used to to provide enforcement of policies in a federated military network. To understand this question it is reasonable to break it down into smaller pieces:

– What inherent problems do we face with policy enforcement in federated military networks?

– Can SDN be used to solve these challenges? And if so, in what way?[1]

– How does these solutions compare with legacy solution?

This chapter will elaborate on these questions, provide a discussion on the matter, and ultimately give a conclusion. A subset of this conclusion is meant to be used later in the work on possible designs in chapter 4. A subset of that design solution will then serve as the basis for an implementation of a specific policy rule in chapter 5.

3.1 SDN in a Federated Environment

A federated network consisting of several interconnected network segments (see figure 3.1) from different coalition partners is an important aspect of modern battle space resources. Providing policy enforcement for one segment of such a network is one challenge, but providing coherent policy enforcement for all flows traversing all segments of a federated network is another.

[1]Some of the work presented here is based on the authors own unpublished work in a preliminary project report [42] written before this master thesis.

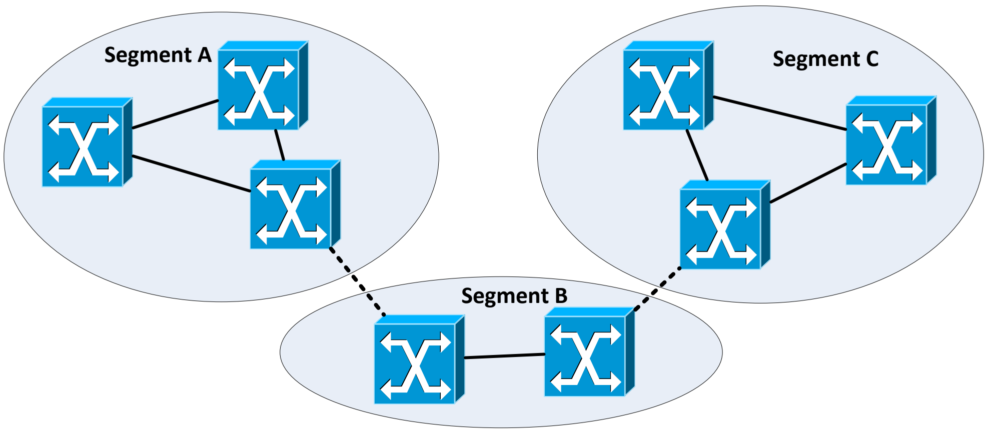


Figure 3.1: Segments in a Federated Network

Policies can in this scenario be thought of as stated rules for how the traffic flows should be handled in the network. Whether the rules are connected to access control, QoS, reliability or otherwise. These rules will first have to be agreed upon between the different network operators (nations) through an SLA or other forms of policy rule lists or definitions.

The first obstacle with policy enforcement in federated networks is a high level agreement of the actual policy rules. This is more of a political challenge when it comes to military networks. All nations have to come together and agree on which policy rules they want to implement. Then they have to provide trust to one another so that the different operators can be sure that their traffic gets treated correctly, are allocated the proper resources, and is sufficiently secured in other segments of the network than the one(s) they operate and own. This also applies to the degree of dependability that is to be expected from the network. In military networks availability and survivability is considered very important. The operators (nations) have to be sure that if they rely on resources in a different segment of the network, this should always be available.

Federated networks should provide mobility for nodes. I.e. that a host moves through the network, and connects to different parts of it. This is a challenge in traditional networks. In SDN this can be solved through using SDN as an overlay network, as in IBM's solutions presented in [43]. This way you could support seamless relocation of hosts through the federated network, which is in compliance with the guiding principles of PCN (ref. table 2.3, *Support of Dynamic and Federated Environments*).

Another challenge with federated networks are the way management and control is performed. I.e. how policy rules are implemented globally across the network. In legacy networking the control logic for implementing policy rules on the network is placed on the individual network devices (switches) or higher up in a closed vendor specific network stack (i.e. a vendor specific management system). In an SDN enabled network, management of network policies can be placed in a policy engine application at the centralized controller (controller or application plane). However, this raises a question for federated networks of where the controller should be placed. A few different solutions can be assumed:

1. Distributed controllers, one for each segment (ref. figure 3.2).
2. Hierarchically tree structure with a central top level controller (ref. figure 3.3).
3. One central controller for all segments (ref. figure 3.4).

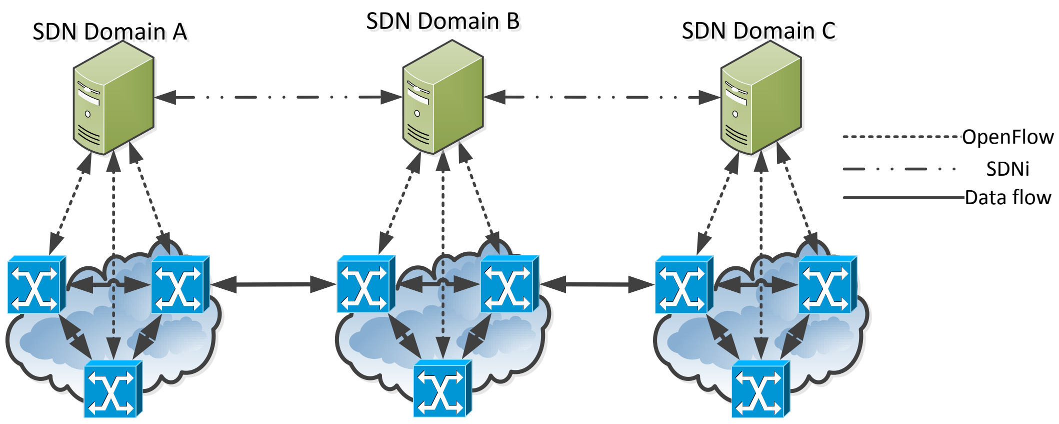


Figure 3.2: Distributed Controllers

With distributed controllers you would have to use some sort of east/west communication protocol between the controllers to share topology information etc. The IETF has proposed a protocol for this called SDNi[44], but it is still only in draft form. The SDNi protocol is however only relevant if you need a standardized east/west protocol. Specialized protocols could be written to function between controllers of the same type. This is much of the same discussion as with standardized northbound APIs (ref. chapter 2.1.2). The control of the network would here be distributed over a set of controllers, comparable to what you would see with distributed routing protocols. This means that no single (centralized) entity has a ”true” view of the complete network. Each controller would have to send their view of the network to the other controllers to be aggregated each time there is a topology change on any of the segments. This implies a flooding of information from one to all, which will mean a lot of signaling traffic have to travel between the controllers. It also implies that when a new path (or set of flow rules) for traffic traversing several segments is computed in one of the controllers, there have to be some sort of mechanism that prevents the insertion of flow rules if the topology has changed during the course of the path computation (e.g. as in MPLS-TE (ref. chapter 2.3) where RSVP-TE is used). For instance, assume that controller A receives topology updates from controller B. Controller A wants to create a tunnel that uses resources in segment B. It creates rules based on its current topology understanding of segment B. Before the new flow rule installation request reaches controller B, the topology has changed, invalidating the new rules. Controller B, which has the correct topology view, would then have to reject these rules based on a comparison with its current topology view. A negative response should then be sent from controller B to controller A. An updated topology view should be sent at the same time.

The previous example raises some core questions concerning the usage of several controllers. Especially if these are operated by different entities, and not one as in the B4 SDN usage case (ref. chapter 2.1.4). Namely, what sort of topology information should be shared between the controllers; a global topology view of that segment, or just a subset. And to which degree should controllers from other segments be able to control network devices in another segment (i.e. push new rules); full access, or just to a designated part (a slice) of the resources. Full access, and full topology view seems like an unlikely and less than optimal solution. Especially since it's highly unlikely that network operators (nations) would like to share all information about their network with others (from a security point of view). A far better and more manageable solution would be that controllers share a subset of their topology view with neighboring controllers, and receive requests for flow setups in return. These requests could then be compared with the current topology view of that segment, before being translated into low level flow rules and forwarded to the appropriate devices. In a federated network it does not seem reasonable to share a full topology with all other segment. The reason being that some resources will probably never be shared with other segments. Sharing the complete view would only cause unwanted computational burden on both controllers. This can be further underlined when compared to the findings in [45]. Here the authors argue that the large number of OF specific microflows between controller and devices will not scale for large high performance networks. If the controllers also have to flood all topology updates to other controllers in the federated network, this problem will only grow. An aggregated subset of the most important updates (e.g. only of the resources you would like to share) would be a more prudent solution.

When it comes to the control aspect of the actual devices, it also seems unlikely that neighboring controllers should be given full access to push new flow rules into devices on other segments. A better solution would be to send requests in form of policy rules, and have the local controller translate these into actual flow rules. This gives the operator of a segment the possibility of moving tunnels from other segments around more freely (as long as they are still in compliance with the agreed terms, policies, SLA etc.). This could prove especially valuable in the case of pre planned maintenance. E.g. if a node have to be upgraded or moved.

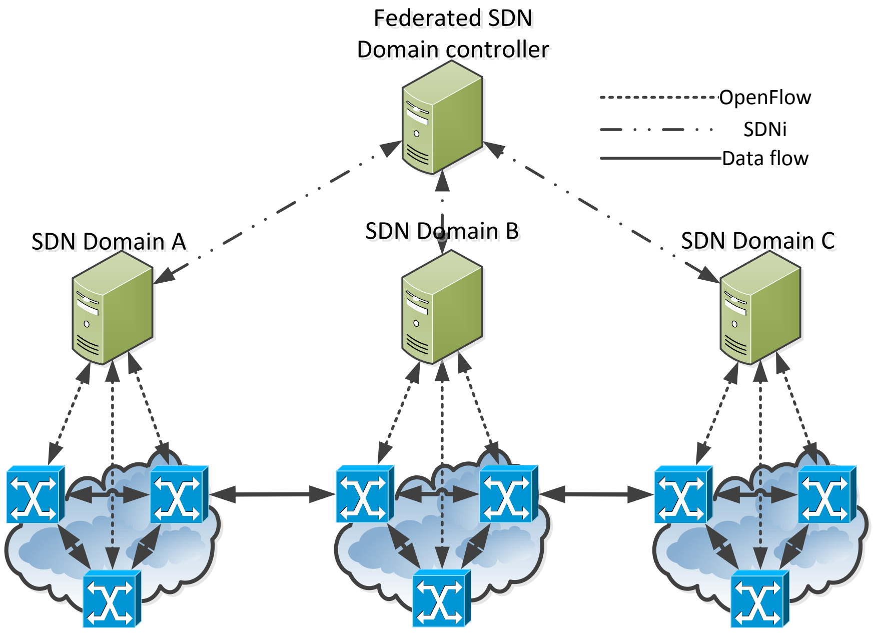


Figure 3.3: Hierarchical Controller

With a Federated SDN Domain Controller (FSDC) placed hierarchically atop of the SDN segment controllers you would have one centralized place where the topology information is shared for the whole federated network, and where path decision are created for all paths traversing several segments of the federated network. The top controller would pull and receive topology information for all other controllers. Calculate network paths, and send flow rules to the underlying controllers for each segment. This can in some sense be compared to Google's B4 internal WAN network [4] where they have one centralized TE server that does traffic engineering for the whole network, and one network controller for each data center (which can be viewed as a network segment). The traffic engineering node in Google's case does also support edge rate limiting at the application level. This rate limiting serves a large part in how they are able to achieve close to 100% network utilization. I.e. when the network is reaching its threshold of available resources, low priority applications can be rate limited at the edges, thus keeping the packet loss at a minimum. One could envision the same practice in a federated military network, but it seems unlikely. This is because a central shared node would have to have control of edge rate limiting for all applications in all segments (nations). This is the same argument as used earlier, concerning full sharing of topology information.

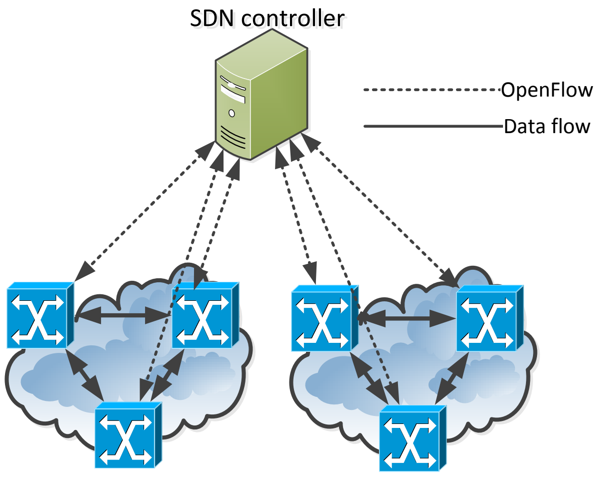


Figure 3.4: Single Controller for all Segments

With only one central controller for all segments (ref. figure 3.4) you will not get any inconsistencies in topology view, as you might get when you have several controllers in a distributed fashion. However, using only one controller have several drawbacks. The first and largest concern is dependability issues. If the central controller should fail, the whole network will in effect be failed. This is unless the switches run in hybrid mode, where a local control plane on the devices can process new flows, or the controller have a hot stand-by. The latter will however not have any effect if the network links between one segment or more and the controller fails. With only one controller you will also have a long flow rule installation time because of accumulated latency times when a new flow have to be sent back and forth between the network device and the controller. A problem assumed to be one of the larger challenges when it comes to scalability in SDN. For a large federated network, the computational load could also be to high for a single controller to manage efficiently. It is also highly unlikely that different nations of a federated military network would accept a single controller to have full control of all their infrastructure. Especially if it's placed in a different location than the main bulk of their segment. Both from a security and survivability point of view.

A single controller architecture is an unrealistic solution for implementation in a federated military network.

Considering the three different architectures for controller placement, it seems favorable to use hierarchical controllers (ref. figure 3.3) in a moderate form. Where the top controller is used for traffic engineering and policy enforcement for parts of the different segments that each nations chooses to make available. Such as controlling and creating tunnels for *local coalition traffic* (ref. chapter 2.5). One could also envision the top controller providing edge rate limiting for applications and services that are used to share information between the different nations. I.e. applications that are provided by the coalition as a whole.

3.2 Relevant Policy Categories

As an uptake for the design and implementation part of this thesis we have pin pointed some general policy categories that are relevant for federated networks. These are also viewed as possible to implement in a minimal test bed. The categories chosen are related to how the traffic is steered, and what kind of service (QoS) it receives. Other policy types, such as types that govern access control and security is regarded out of scope.

3.2.1 Category 1: Best Effort and Robustness

This policy category includes policies that moves the network between degrees of best effort and robustness. Best effort is a service model in data networks where there are no QoS commitments of any kind [46]. I.e. all flows in the network are treated equally, with no guarantees for delivery. This creates a scenario with a good probability of high utilization, but it also means that the network can easily become congested. E.g. a makro flow taking up all bandwidth, and therefore blocking all micro flows. Different service models for a data network can be viewed as a vector that goes from best effort to robustness (ref. figure 3.5).

Best Effort
Robustness

Figure 3.5: Vector on Best Effort and Robustness

A network with a high degree of robustness could for instance load balance traffic over several paths or implement more robust routing protocols that are able to continue packet forwarding faster [47]. A higher degree of reliability will create a more robust network, but all reliability schemes will in some part lower the utilization of network resources. Especially if backup paths are idle, or reserved bandwidth is underutilized by the allocated traffic type. The vector depicting the relationship between best effort and robustness in data networks can therefore be viewed as having several *shades of grey* between best effort and ”maximum” robustness. Consider a dual link between two nodes. One scheme for implementing robustness in this network could for instance be to use a 1+1 masking protection scheme [48]. This will create a network that is quick to recover from a link failure, but it will result in a 50% utilization degradation of the available capacity.

For federated military network there will very seldom be room for best effort only. Different services and applications have to be prioritized, and some have to be given higher reliability. E.g. a Blue Force (BF) tracking system or a weapons systems connected to a radar or other sensory data sources.

With SDN it is possible to implement different policies that move the network characteristics between best effort and robustness in a dynamic fashion. In this scenario defined as operator induced dynamics, meaning that an operator can easily change how the network operates during an ongoing operation. An example of this could be where groups of flows are given specific QoS tags. With SDN these tags doesn’t have to be attached to the header values. They could rather be stored in a database at the controller:

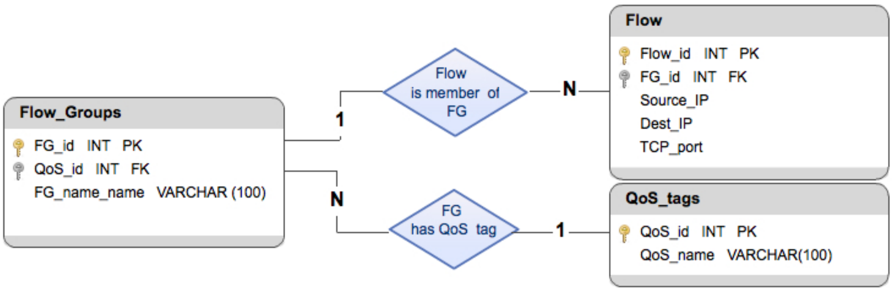


Figure 3.6: Database Diagram for QoS Tags

Figure 3.6 is just a simple mock-up diagram for a possible entity relationship. An actual implementation would probably look quite different. The point being, that the tables stored in a local controller database could be used to organize different flows as the operator wants. An actual QoS tag doesn’t have to be added to the flows. Although, you could use OF actions to push a new header with a tag onto the packet. If a tag isn’t added to the packet, you would need some other mechanism for providing QoS service throughout a federated network. Such as having a shared database with flow groups and QoS tags for all segments of the network. Using the above mentioned method with a local database, an operator could create a network where he uses different methods[2] to create paths through the network

[2]The term methods is here meant to describe programmed methods in an application on the controller that can be invoked to create different flow rules.

for flows in different FGs or with different QoS tags. You could for instance have methods for creating SPF or Constrained Shortest Path First (CSPF) paths, or for adding a flow to a group table of one node that will send the flow onto the first live port in a group of ports (for increased dependability). If an operation changes, paths can then be changed for a number of FGs (consisting of several flows) quickly. Antoher possible solution here is to be able to downgrade several FGs through assigning them to a QoS tag with lower priority through the network. E.g. to put all flows from nation B into the same flow group, and then change traffic characteristics for all these using their common FG id as input to a method. This gives a flexibility and speed to the management of a network that is not seen in legacy solutions such as MPLS-TE.

3.2.2 Category 2: Allocate Network Resources

This policy category includes policies that allocate and re-allocate network resources. I.e. moving paths or flows around the network. This would be an integral part of a traffic engineering solution where the object is to utilize the available resources in the best possible way. The thought behind this policy class is to provide policies that enable the network operator the possibility to share and reclaim network resources easily as changes occur. This is an important concept in federated military networks, and stated as such in PCN [39, §2.2], as it's the commanders prerogative to be able to command all resources under his or her command. This concept is often overlooked when it comes to network resources, as traditional military resources most often is in focus.

Assume a federated network with segments from different nations. Say for instance that nation A has available bandwidth. It can then allocate this to Nation B if needed. Either per output port (link), or per queue on an specific output port[3]. If the path for nation B's traffic is later needed by nation A (the owner of the network), the established paths have to be moved. This should be done without much packet loss or downtime. To create a solution for this in SDN you would have to create an application at the application plane, or implement the same functionality as a core service/module in the controller itself. The latter being the strongest solution as it can reuse functionality that's already in the controller, and it will operate faster with a more direct access to southbound interfaces and other core services. The following figure shows a possible architecture for this:

[3]The differentiation between output queues were implemented in version 1.0 of OF as support of multiple queues per output port [20, pp. 150]. This is known as *slicing* in the OF specification, as you can provide a "slice" of the total bandwidth of the output port to each queue.

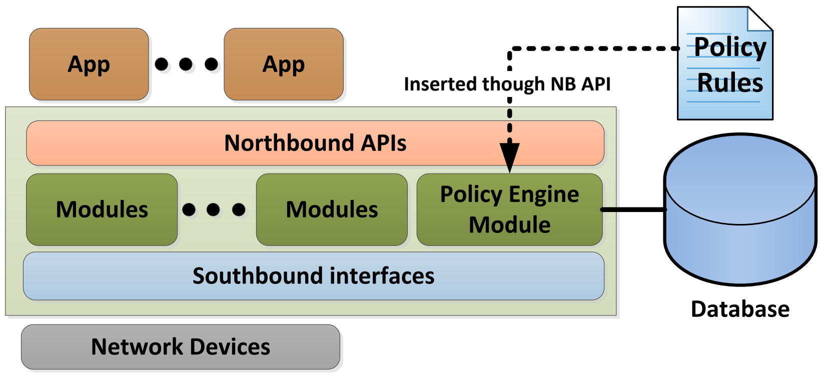


Figure 3.7: Internal SDN Policy Engine

Figure 3.7 doesn't take communication between neighboring controllers into consideration. This is in correspondence with the scope set for this thesis, as mentioned earlier.

The solution showed in figure 3.7 assumes the implementation of a policy engine module in the control plane. Stated policy rules are inserted using a northbound API. A Database (DB) is used to store flow rules connected to different flows, flow groups, etc. If a flow has to be moved, a policy will be changed. The policy engine will then check the database to see which relationships apply. Storing information about flow rules in a local database will prevent the controller (policy engine) from having to search through all network devices for rules that have to be changed. This implies a reduction in signaling traffic between the controller and devices.

3.3 Partial Conclusions

From the discussion we can see that there are several challenges concerning policy enforcement in a federated military network. From a political perspective we have found challenges connected with agreements between operators (nations) on global policies and the amount of topology information that should be shared. We assume that these are possible to solve through a discussion between the different nations. At the same time, most of the topology information doesn't have to be shared between different nations, only an aggregated subset of the resources they want to share (e.g. underutilized links).

Connected with this it the need for nations to receive the proper amount of reliability and service for traffic transported through other segments of a federated network. This is also assumed solved through SLAs. At the same time, nations should not lose control of the physical infrastructure in their own segments when sharing resources.

Other challenges with federated networks are coherent policy enforcement for all segments, and management of network resources.

While some of these challenges could be regarded as purely political, and mostly connected with creating good SLAs and high level policy rules, we see some areas where SDN can add value. Especially through the flexibility in network management that SDN can provide. With a sound implementation of a policy engine in the controller, policy enforcement is possible through the use of SDN. The use of SDN can also provide the flexibility to implement novel management methods and policy rules that fit specific network scenarios. Management systems that don't rely on closed vendor specific solutions or configuration of devices one by one can also be developed for use in SDN.

Node mobility is also something SDN can provide, and which could prove useful in a federated network setting.

There are however some weaknesses with using SDN at the current time. Inter-domain state distribution and east/west communication protocols (e.g. SDNi) are scarcely researched, also in this thesis. However, one thing is clear: If SDN is to be used in a federated network, a communication protocol have to be implemented between the different controllers. At least between segment controllers and a hierarchical top controller (ref. figure 3.3). A scenario closely related to the architecture of B4 [4].

Another weakness with SDN is dependability issues with using a central controller. Dependability schemes using fail-over controllers and hot-standbys should be studied further, as well as scenarios with link failures between controller and network devices. The latter seems especially important due to the dynamic and harsh nature of military operations.

Both of the first network structures discussed in section 3.1 of this chapter (ref. figure 3.2 and 3.3) relies heavily on the use of east/west protocol between the different controllers. Implementing such a protocol would require a lot of work as it's not readily available. It is therefore considered to be out of scope for this thesis, with regards to the design and test bed implementation work. However, as a proof of concept for using SDN in a federated network, an implementation of a single controller architecture in the test bed will still be a valid solution. A network topology with a single controller will therefore be used in the design and implementation phase of this thesis.

Chapter 4

Design

This chapter looks into possible designs for policy enforcement and policy rules in an SDN enabled federated network. The scenarios are kept simple to give an overview of what could be accomplished through the use of SDN. The designs are predominantly based around scenarios where policy changes have to implemented. I.e. where the state of the network changes.

4.1 Drop Low-Priority Packets

This section follows the policy category described in section 3.2.1. This design is meant to provide more robustness in a network for high priority traffic flows.

This policy could prove to be important in a general military network where the topology is dynamic and you have heterogeneous network bearers. The main idea in this design concept is that you have a network with two different network links reaching back to a backbone network. For instance a SATCOM link with a high data rate and a radio link with a low data rate. If the SATCOM link goes down, the network should react accordingly and only route high-priority time critical flows over the low data rate radio link. The topology described in this design scenario is depicted in the following figure (ref. figure 4.1):

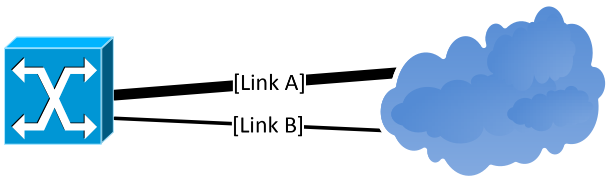


Figure 4.1: Multi Path Reachback Network

In this topology the switch is considered to be part of a federated network where nation A shares resources on the two links with nation B. One is a high rate link ([Link A]), the other is a low rate link ([Link B]) with less bandwidth and more latency. In this scenario link A goes down more often than link B[1]. Both links are reachback to a backbone network which can support all nations in the federated network with some service. In this scenario a critical BF tracking system[2]. With this system it's imperative with a high degree if availability. Consider that the network is located in a Command Post (CP) which is built on an Armored Personnel Carrier (APC) platform. The CP aggregates a blue force view from the units on the ground and sends this information to the backbone. This information is then pumped into an air force system so that pilots can see where friendly ground forces are located. The size of the data packets from this system is considered to be small.

The policy in this network should operate so that when the CP is in a static position (i.e. Link A is up) the network traffic is given a best effort service. Low priority traffic, such as mail and video streams, are sent on this link together with the high priority blue force traffic. When the CP is mobile and link A is down only BF traffic should be able to pass over the remaining link B. All other traffic should be dropped. Also, when in a static position and both links are up, BF traffic should be routed onto the best alive link, with a minimum bandwidth guarantee.

This could be implemented with the use of SDN in the following way:

1. A group table forwards traffic on the first live port in a group of ports - Match with UDP[3] port number for BF flows
2. Set up flow forwarding for all other traffic with just [Link A] as output port.
3. Rate limit all non-BF traffic with a meter table.

4.2 Re-assign Tunnels

This design follows the policy category described in section 3.2.2. This design is meant to provide a solution to move tunnels from one path to another.

One possible scenario[4] for policy enforcement in a federated network is where you have to reassign network resources (given to another nation) on your own network. This could

[1]The reason for this could for instance be that link B has more mobility than link A. A rather common scenario in military networks where better uplinks are set up when the operation is static, and worse links are used when on the move.

[2]Blue force tracking systems are a system that interconnects military units and share position updates to avoid friendly fire accidents.

[3]UDP is more commonly used in BF systems than TCP.

[4]In this scenario, *slicing* is not taken into account. Each part of the federated network is assumed to be under control by a separate SDN controller from each nation. Traffic from the other nation enters the network in question on node R1, which could be viewed as nothing more than a gateway to some link on R5.

for instance happen when spare resources (on a network link) are allocated in the form of a tunnel, but where changes in an ongoing operation imposes that you need to take the resource back. In this scenario an important factor is that you don't remove the allocated tunnel completely, but rather moves the tunnel to another path through your network. Traffic flowing through the tunnel should still be able to pass, even though the latency or bandwidth might be degraded. The following multi path network topology between node R1 and R5 is used as an example (see figure 4.2).

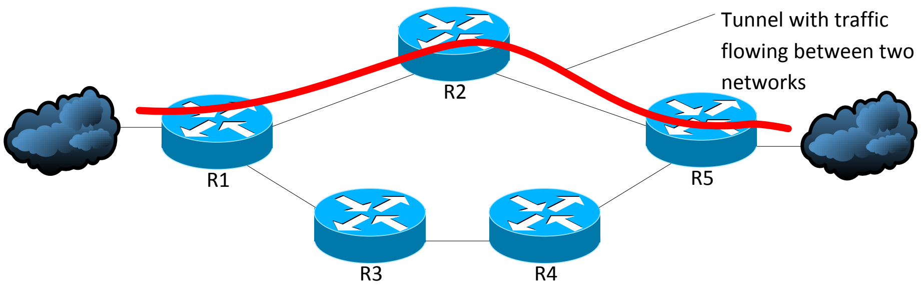


Figure 4.2: Simple Topology Where an Established Tunnel Have to be Moved

In this topology the tunnel is firstly placed on the path $[R1\text{-}R2\text{-}R5]_{100Mb/s}$. This is a path with fewer hops and higher bandwidth then the path over path $[R1\text{-}R3\text{-}R4\text{-}R5]_{10Mbp/s}$, and were therefore the path first chosen to allocate the tunnel to because it had a significant under provisioning of available resources. Then the ongoing operation changes, and the path have to be freed for use by the nation that owns this segment.

4.2.1 Solution in SDN

In SDN, the task of accomplishing this scenario can be broken down to a few main parts:

1. Create the original tunnel using SPF or other path computational algorithm:
   - Create and add the flow rule to R1, R2 and R5 (bi-directional). Store the flow name
   - Add the flow to a meter table with the appropriate rate limiter
2. Use a move method† to move the flow from $[R1\text{-}R2\text{-}R5]_{100Mb/s}$ to $[R1\text{-}R3\text{-}R4\text{-}R5]_{10Mbp/s}$:
   - Create and add new flow rules to R1, R3, R4 and R5 (bi-directional)
   - Read back the old flow rule name and delete these together with the old meter table entries

- Add the new flow to a meter table with the appropriate rate limiter

The move method (†) is here considered the actual implementation of the policy change. As this rule differs from the normal SPF path creation, it's thought as a way to easily move paths on the fly using a set of high level commands in the application plane. This means that these type of policy changes can be made almost instantaneous without the need to configure each network device manually for each flow. Of course, this implies that a policy/network management system fulfilling the needs of such commands have to be coded and implemented before hand. A typical work flow for this kind of policy management could be the following:

1. New tunnel creation: Add a SPF tunnel with match fields for IPv4-destination, and with action fields for output-port and meter table with rate limiter.

2. The need of the network changes: Find out what this means for the traffic in network, pin-point the tunnels that have to be moved.

3. Move tunnel: Move all tunnels ”manually”. This could for instance be done through invoking a move method with two inputs: A list of the tunnel names that should be moved (i.e. deleted and recreated) and an excluded node which shall not be used in the path calculation.

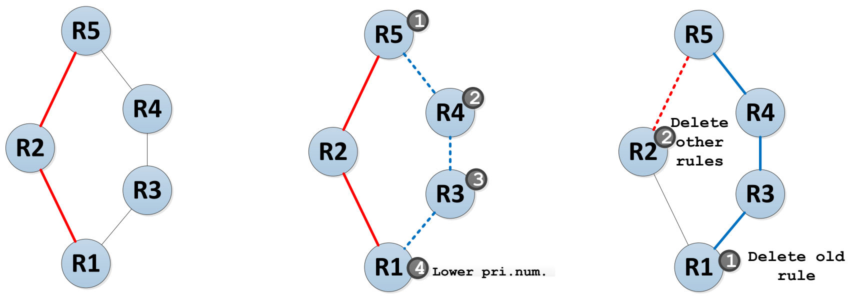
   - The move method will then take these inputs, iterate over them and create the new tunnels in one swift operation.

An example of the move method/function could for instance be (in pseudo-code):

```
function move(listTunnelNames, excludedNode):
  for element in listTunnelNames:
    delete flowsrules where flowname == element
  listOfNodes = calculateSPF(graphOfTopology, excludeNode)
  for element in listTunnelNames:
    create flowrule with flowname = element
    for element in listNodes:
      PUT newflowrule to element
```

Listing 4.1: Move Function in Pseudo-code

The move method should also follow a specified order/sequence for the flow rule installation and deletion (not considered in listing 4.1). When new rules are created they should be pushed in order, following the path from destination to source (relative to one direction of a bidirectional path at a time). When installing the flow rule on the node that has the first split between two parts of a multipath (node R1 in figure 4.2) it should insert the



(a) Original tunnel. (b) New tunnel unused. (c) New tunnel in use.

Figure 4.3: Description for Move Method. Only for Traffic Moving from Source R1 to Destination R5

new rule with a lower priority number[5] for that flow rule before deleting the old flow rule. This way, the traffic flow will continue to function without packet loss or loop possibilities. All other flow rules for the original tunnel should then be deleted. Flow rules installed on nodes before the multipath split should then be updated in dest-src direction one by one if for instance the rate-limits on meter tables or similar have changed. If they are correct no action have to be taken.

4.2.2 Comparison: Cisco MPLS-TE

If the same task should be done through the use of MPLS-TE on Cisco routers the following steps would have to be taken. This is an explanation of how it's done based on configuration commands in the IOS of a Cisco MPLS-TE enabled router. Assume that the network is running the OSPF routing protocol. The original *Tunnel1*[6] is set up pretty automatically using the OSPF protocol, even though some configuration have to entered (excerpt):

```
tunnel destination 11.0.0.1
tunnel mode mpls traffic-eng # To enable MPLS-TE
tunnel mpls traffic-eng autoroute announce # Tunnel will be
   announced by the routing protocol
tunnel mpls traffic-eng bandwidth 1 # To enable RSVP
```

A routing protocol also have to be set up for the MPLS network, in this scenario OSPF is used:

[5]OpenFlow uses a priority number for matching precedence. See [20, § 5] for reference, and listing 5.2 in chapter 5.1.1 for example

[6]$[\text{R1-R2-R5}]_{100\text{Mb/s}}$

```
    mpls traffic-eng area 1
    mpls traffic-eng router-id Loopback1
```

This will (in short) create Tunnel1. If this then have to be moved, now called *Tunnel2*[7], two possible methods are available:

1. Configure an explicit path, where the IP addresses of all nodes the tunnel are explicitly mentioned:

   ```
   ip explicit-path name Tunnel2 enable
     next-address <ip-address for R3>
     next-address <ip-address for R4>
     next-address <ip-address for R5>
   ```

2. Exclude R2 from the PCE calculation:

   ```
   ip explicit-path name Tunnel2 enable
     exclude-address <ip-address for R2>
   ```

This is pretty straight-forward, and is easy to do if you only have to move one tunnel. On the other hand, if you want to reassign several tunnels to a new specific/explicit path it becomes more cumbersome. In that sense, it will be easier to do the same operation using SDN. This is of course under the assumption that the framework is in place to do this operation. Also, MPLS-TE can only be used on MPLS-TE capable routers, which causes hardware dependencies. With SDN, this specific scenario could be created using any OF capable multi-port network device (switch or router).

4.3 Chosen Design for Implementation

When choosing which designs to implement we looked at relevance of the design. As resource sharing is an important topic for military federated network, the design for re-assigning tunnels were chosen to be used in the implementation test bed.

The following table (ref. table 4.1) lists a number of ambitions for the implementation work, and what we wanted to achieve through the the test bed:

[7] $[\text{R1-R3-R4-R5}]_{\text{100Mb/s}}$

Part	**Description**
SDN controller	Research different SDN controllers and find one fulfilling the needs of the implementation.
Network Emulation	Establish a functioning network emulation environment that can be used to validate the functionality of the implementation.
Policy application	Write an application at the control plane that are able to: Receive and update policies. Utilize the available resources through the use of multi path. Install flow rules reactively and proactively.
Reactive rule installation	The application should be able to install rules automatically when an unknown packet (with non-matching match fields) arrives to the network, following a set of policy rules. E.g. that all packets from the following subnet should use a defined path.
Proactive rule installation	The application should be able to install rules proactively
Dynamic Enforcement	The application should be able to respond to sudden changes in network topology, e.g. link failures.

Table 4.1: Ambitions for the Implementation

Chapter 5

Implementation

During the work on this thesis we created a test bed for policy enforcement in a federated network, using SDN principles and the OF protocol. The work on this test bed enhanced our understanding, and showed clearer the possibilities, constraints and challenges connected with the use of SDN. This was an important, and large, phase of the work as the authors understanding of SDN were mainly based in theory, and not the practical steps associated with an SDN environment. SDN, and the use of the OF protocol, seems easy on ”paper”. But, as the work on the implementation showed this is in no way easy and straightforward. The following sections of this chapter are the steps taken to research, set up and develop on the SDN platform.

5.1 Possible SDN Controllers

When choosing a controller for the test bed, a number of different implementations were considered. Several different controllers have been created by the industry and in academia. Among the most important aspects considered were:

– Features and support for needed functionality

– Familiarity with programming language

– Level of documentation

– Size of user community

From this work, a number of controllers were identified. The following were considered (ref. table 5.1):

Name	**Description**
OpenDaylight [21] [49]	Written in Java, runs within a JVM on the OSGi framework. Has received a lot of attention from the community. Open source software with several large companies backing the project with money and developers. Has a vibrant development community with an active IRC channel. Supports a large number of southbound interfaces including OF 1.3, and has a northbound REST [50, Ch. 5] interfaces to most of the functionality.
NOX [51] [52]	Written in C++. One of the earliest SDN controllers, built at the same time as the initial OF version. Has later served as a basis for other SDN projects.
POX [53] [54]	Based on the NOX SDN controller. Written in python, and supports OF version 1.0. Very little development on the controller as of late. Has seen wide use in research and academia, but spread of the controller seems to be stagnant.
Beacon [55] [56]	Written in Java. Developed at Standford University. Has not seen a lot of new development during the last year. Runs as a stable controller for the Stanford SDN network.
Floodlight [57] [58]	Written in Java, runs within a JVM. The project is supported by the company ”Big Switch Networks”, and is a fork out from the Beacon project.
Ryu [59]	Written in Erlang. Most of the developers are Japanese, and very little documentation are available. Does not seem to have a large following or community but has seen some use in academia.

Table 5.1: Available SDN controllers

When considering these controllers we put special emphasis on the level of ongoing development and user community. An example of this is the POX controller. This has widespread use in academia and many users, but hadn't been updated for eight months when controllers were considered. Also, the lack of OF 1.3 support were disqualifying. After considering the different identified controllers (and trying some of them), ODL were considered as the most promising candidate.

5.1.1 OpenDaylight

The ODL controller were under active development when it was chosen, and it was not formally released[1]. Even though the software were quite immature and untested (from an application development stand point), it had a vibrant community of developers. The devel-

[1]ODL were released february 5. 2014 [60]

opers were also easy to contact directly as they all communicate on a public Internet Relay Chat (IRC) channel[2]. A wiki-page were also actively update with controller information, user/development guides, tutorials and API references.

The ODL project is an open source software project that seeks to create an SDN controller that can be used as a baseline for SDN development. Several large network and software vendors contribute to the project with code, developers and financing. Among the largest contributors are such companies as Cisco and IBM.

The architecture of ODL can be viewed in figure 5.1:

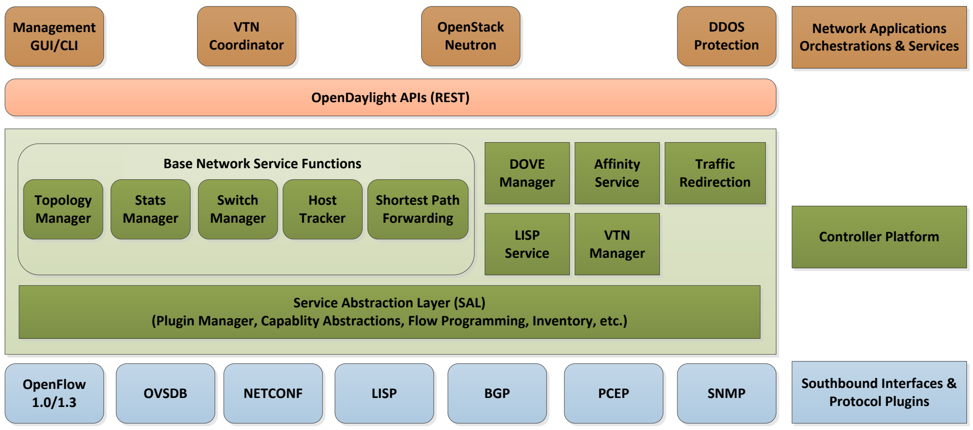


Figure 5.1: ODL architecture [61]

The ODL controller runs in a Java Virtual Machine (JVM) with the Open Service Gateway initiative (OSGi) framework [62]. Using the OSGi framework makes it possible to add, remove and update bundles/modules without having to reboot the complete system. The modules of ODL can be viewed in figure 5.1 as ”green boxes” above the Service Abstraction Layer (SAL) in the controller platform.

From the ODL controller there are a number of APIs available. These are basically divided in two. Layer 2 (controller plane) APIs that are internal to the controller, and who works between the different bundles, and layer 3 (application plane) APIs that are north bound from the controller.

[2]#opendaylight channel on irc.freenode.net

Service Abstraction Layer APIs

The internal APIs (layer 2) are created for communication between the different modules in the controller

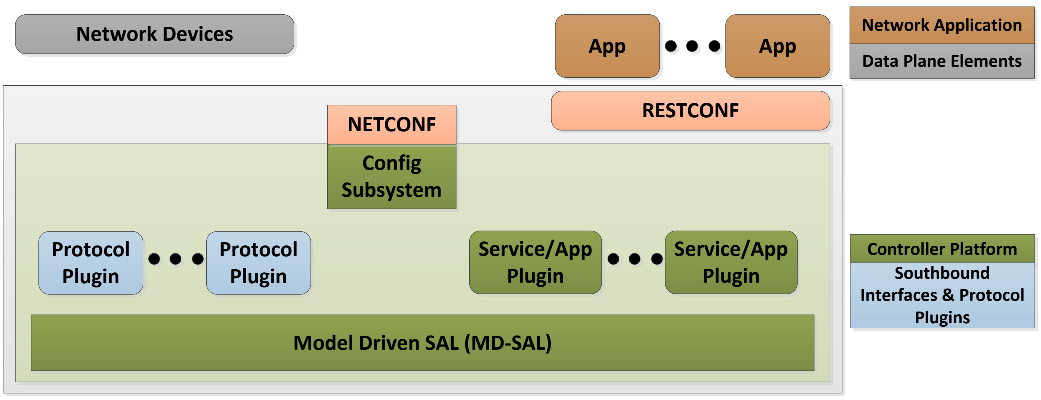


Figure 5.2: MD-SAL in OpenDaylight, from [63]

As seen in figure 5.2, the modules in ODL are connected using a shared message bus called Model-Driven Service Abstraction Layer (MD-SAL)[63] (This figure should be viewed in comparison with figure 5.1 for better understanding). As seen in the figure, from a software development view, MD-SAL sees no difference between *southbound* or *northbound* interfaces. When creating a controller application (e.g. a policy engine), the most powerful solution will be to place it as a module in ODL within the OSGi framework. This will give the application module access to the other modules in a direct way, and it can be both a consumer and provider of data within the controller. An example of this is a simple learning switch application that's bundled with the ODL release. This gives learning switch functionality to the network so that you can run a flat LAN over your network devices. It emulates common functionality found in L2 switching such as Address Resolution Protocol (ARP) and Spanning Tree Protocol (STP). As the Learning switch application is placed in the controller plane, and connected to the MD-SAL, it can easily consume *packet_in* messages that arrives from the network devices to the OF plugin. It can also produce data that the OF plugin consumes for encapsulation into *packet_out* messages towards the network devices. This functionality is not available through the northbound REST APIs.

Nortbound REST APIs

The north bound (layer 3) REST APIs (also called RESTCONF, ref. figure 5.2) are easy to access through HTTP requests from most programming languages or applications. Most of the bundles in the controller have north bound REST APIs implemented. They give a basic functionality for communicating with the controller. A full reference for the different REST APIs can be found at [64].

The REST APIs in ODL are based on the concepts of REST[50] and provides namespace Uniform Resource Names (URNs) towards the different modules/bundles and components of the controller. This system is based on requests from the northbound application (client side) towards the controller (server side) where Extensible Markup Language (XML) data bodies are sent and received. Most of the information that the controller stores about the network can be received and changed in this way. For instance, if a northbound application wants to add a flow to one of the switches it will first request data from the controller about the topology with a GET request towards the topology namespace (e.g. http://localhost:8080/restconf/operational/network-topology:network-topology/). The controller will then respond with an XML body containing all network topology information, for instance (excerpt):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<topology
    xmlns="urn:TBD:params:xml:ns:yang:network-topology">
    <topology-id>flow:1</topology-id> %ID of the network topology
    <node>
        <node-id>openflow:1</node-id>%Name of the network device
        <inventory-node-ref
            xmlns:tyhj="urn:opendaylight:inventory"
            xmlns="urn:opendaylight:model:topology:inventory">
            /tyhj:nodes/tyhj:node[tyhj:id='openflow:1']
        </inventory-node-ref>
    </node>
    <link>%One link in the network
        <destination>%Name and portnumber for destination node
            <dest-tp>openflow:2:1</dest-tp>
            <dest-node>openflow:2</dest-node>
        </destination>
        <link-id>openflow:3:3</link-id>%Name for this link
        <source>%Name and portnumber for source node
            <source-tp>openflow:3:3</source-tp>
            <source-node>openflow:3</source-node>
        </source>
    </link>
```

Listing 5.1: XML Topology Response

This basic information can then be used by the application to find out on which nodes it want to add flow rules. The application will then build flow rules and send these to the controller using a PUT request. The controller will then push the flow rules to the corresponding network devices (switches). A flow rule could for instance be on the following form:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow xmlns="urn:opendaylight:flow:inventory">
    <strict>false</strict>
    <flow-name>Flowrule1</flow-name>%Unique name of flow on that switch
    <id>258</id>%ID of flow on that switch
    <cookie_mask>255</cookie_mask>
    <cookie>103</cookie>
    <table_id>2</table_id>%ID of the table in the pipeline that stores that
         flow
    <priority>2</priority>%Priority number for that flow [1:500]
    <hard-timeout>1200</hard-timeout>%Timeout for when the rule expires
    <idle-timeout>3400</idle-timeout>%Timeout if the rule is unused
    <installHw>true</installHw>%Install the rule in switch directly or
        store at controller
    <instructions>%Start of action fields
        <instruction>
            <order>0</order>
            <apply-actions>
                <action>
                    <order>0</order>
                    <output-action>%A forward action to output port
                        <output-node-connector>1</output-node-connector>
                        <max-length>60</max-length>
                    </output-action>
                </action>
            </apply-actions>
        </instruction>
    </instructions>
    <match>%Start of match fields
        <ipv4-source>10.0.0.1</ipv4-source>
    </match>
</flow>
```

Listing 5.2: XML Flow Rule

This particular flow rule would be sent towards the following namespace: http://localhost:8080/restconf/config/opendaylight-inventory:nodes/node/1/table/2/flow/258. So, the table and flow id's have to correspond in both the XML body and URN namespace. If we consider that this is the only flow rule on that particular switch it would perform a very simple match and action operation where all packets that's received from host with IP address 10.0.0.1 will be forwarded out to port 1. All other packets will be dropped unless other rules are added to the switch by the controller.

What seems to be most important in this thesis work is to understand the possibilities found in the REST APIs towards the OF version 1.3 plugin. This interface can be broken into four main groups: Inventory, Flows, Meters and Groups.

GET requests towards the inventory will retrieve a list of OF nodes that the controller know about (nodes connected to the controller). A request towards the URN namespace http://localhost:8080/restconf/operational/opendaylight-inventory:nodes/node/openflow:1/ will for instance give the following response in XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<nodes
    xmlns="urn:opendaylight:inventory">
    <node>
        <serial-number
            xmlns="urn:opendaylight:flow:inventory">None
        </serial-number>
        <hardware
            xmlns="urn:opendaylight:flow:inventory">Open vSwitch%Switch
                model
        </hardware>
        <software
            xmlns="urn:opendaylight:flow:inventory">2.0.0%Switch model
                version
        </software>
        <manufacturer
            xmlns="urn:opendaylight:flow:inventory">Nicira, Inc.%Switch
                manufacturer
        </manufacturer>
        <switch-features
            xmlns="urn:opendaylight:flow:inventory">%List of the switches
                features and capabilities
            <capabilities
                xmlns:flownode="urn:opendaylight:flow:inventory">
                    flownode:flow-feature-capability-flow-stats
            </capabilities>
            <capabilities
                xmlns:flownode="urn:opendaylight:flow:inventory">
                    flownode:flow-feature-capability-port-stats
            </capabilities>
            <capabilities
                xmlns:flownode="urn:opendaylight:flow:inventory">
                    flownode:flow-feature-capability-queue-stats
            </capabilities>
            <capabilities
                xmlns:flownode="urn:opendaylight:flow:inventory">
                    flownode:flow-feature-capability-table-stats
            </capabilities>
            <max_buffers>256</max_buffers>
            <max_tables>254</max_tables>%Maximum number of forwarding
                tables
        </switch-features>
        <id>openflow:1</id>%Name of the switch (device)
    </node>
  </nodes>
```

Listing 5.3: XML Inventory Response

This is the information about just one node (namely 'openflow:1'), for information about all connected nodes in the same XML body you can send a request towards http://localhost:8080/restconf/operational/opendaylight-inventory:nodes/ instead.

Flow rule requests corresponds to the possibilities found in the OF version 1.3 specification [20]. Examples can be seen listing 5.2 and in files *actions.txt* and *matches.txt* attached to annex A.

Meter requests updates the meter tables of the OF nodes. An example for this can be the following PUT request towards a meter table, that tells the switch to drop a packet if the defined rate is exceeded:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<meter xmlns="urn:opendaylight:flow:inventory">
    <container-name>Container01</container-name>
    <flags>meter-burst</flags>
    <meter-band-headers>
        <meter-band-header>
            <band-burst-size>444</band-burst-size>%Size of bursts
            <band-id>0</band-id>%ID of that band
            <band-rate>234</band-rate>%The lowest rate at which the band
                applies, in kb/s unless a flag is set to look at packet/s.
            <perc_level>1</perc_level>%By which amount the drop precedence
                of that packet should be increased if the band is exceeded.
            <meter-band-types>%Defines how a packet should be processed
                <flags>ofpmbt_drop</flags>%Drop the packet if meter applies
            </meter-band-types>
        </meter-band-header>
    </meter-band-headers>
    <meter-id>1</meter-id>%ID of this meter
    <meter-name>Meter01</meter-name>%Logical name of this meter
</meter>
```

Listing 5.4: XML Meter Request Example

Groups are another useful table to use. An example PUT request to add a new group rule to a group table of a OF switch could be:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<group xmlns="urn:opendaylight:flow:inventory">
    <group-type>group-all</group-type>
    <buckets>
        <bucket>
            <action>
                <set-field>%Rewrites the IP source address
                    <ipv4-source>10.0.0.1</ipv4-source>
                </set-field>
                <order>0</order>%Lowest order is done first
            </action>
            <action>
                <set-field>%Rewrites the IP destination address
                    <ipv4-destination>10.0.0.2</ipv4-destination>
                </set-field>
                <order>1</order>
            </action>
            <bucket-id>13</bucket-id>%ID for this bucket in the group table
            <watch_group>14</watch_group>
            <watch_port>1</watch_port>
        </bucket>
    </buckets>
    <barrier>false</barrier>
    <group-name>ChangeIP</group-name>%Name of the group table rule
    <group-id>1</group-id>%ID for that group table rule
</group>
```

Listing 5.5: XML Group Request Example

In this listing (ref. listing 5.5) we see that the group table applies two actions to the flow sent to it: It uses the optional *set-field* action of OF to rewrite the IPv4 source and destination address. The set-field action is an optional functionality of the OF protocol, and set-field for IP addresses is neither explicitly mentioned in the specification. This is added functionality that the ODL controller implements. This is encouraged in the OF specification as it's said to add *"usefulness [to] an OF implementation"* [20, § 5.12].

Unfortunately, the group table support are currently removed from Open vSwitch (OVS) version 2.0 (which were used in this testbed). The support for this are planned to be added in the next release, version 2.1[3]. This information was discovered too late in the thesis work to make any changes in the test bed environment, so we had to do without this functionality in the testbed[4] (ref. chapter 6.3).

[3]http://comments.gmane.org/gmane.linux.network.openvswitch.general/3251

[4]Note: At a later point we were informed that the CPqD[65] virtual switch does support group tables as specified in the OF 1.3 specification. It is therefore assumed that this could be used before the new release of OVS in future work.

5.2 Network Emulation

At the start we looked into different possibilities for creating a network. We had some hardware switches available (2 pieces), but this didn't seem to cover our needs, with regard to both OF version 1.3 support (the switches support only version 0.9) and quantity. As an absolute minimum, the network had to consist of at least three switches to be able to create realistic scenarios. We started therefore to look into the possibility of using a virtualized switch network.

The emulation of switching networks is an important aspect in the study of computer networks. Solutions to emulate such networks have therefore been developed. This is work mainly done in academia, and they are therefore readily available for use. The most prominent example, and the one chosen in this thesis, is Mininet [66, 67]. Mininet is meant as a tool to prototype networks[5] in a speedy an efficient manner.

Mininet takes advantage of a concept named *process-based virtualization* [66], where *network namespaces* in the Linux operating system (implemented in kernel version 2.2.26) can provide single processes (e.g. Mininet hosts) with their own network interfaces, routing tables and ARP tables. The separate hosts and switches are then interconnected using virtual Ethernet pairs. The following figures is used to explain this in more detail (figure 5.3):

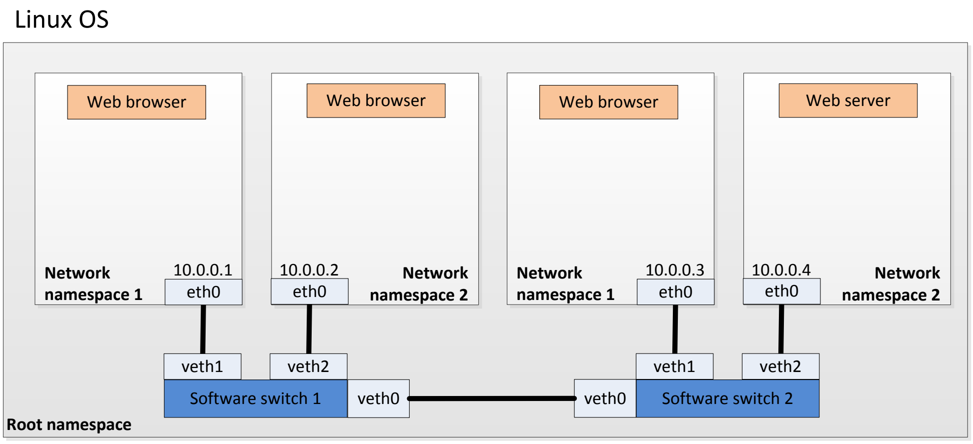


Figure 5.3: Example of a Mininet Topology Running in Linux

[5]If you want to create even larger topologies, Maxinet can be used. Maxinet create large topologies from several Mininet instances spread over several physical hosts. More information on this framework can be found in [68, 69]

In this figure we can see two virtual switches with two hosts connected to each. The hosts are connected to the *vethX* interfaces on the switches. These interfaces can be seen on the host operating system as individual Ethernet ports just like any other port on the host. E.g.:

```
$ ifconfig s1-eth1
s1-eth1    Link encap:Ethernet HWaddr 86:d8:87:0f:ec:e8
           UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:1000
           RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

The OVS type switches that run in Mininet can also be set up to run in hybrid mode where a local control plane on the switch takes over if the controller plane fails (i.e. the controller). This can be with the following command, either through scripting or directly in bash:

```
$ ovs-vsctl set-fail-mode s1 standalone
```

This will set the OVS switch *s1* to operate in standalone mode, which is the same as a hybrid switch as used in OF terminology. An example in scripting can be seen in annex B.

5.3 Detailed Review of the Selected Implementation

This section describes the chosen test bed implementation.

A sketch of the test bed can be seen in the figure 5.4, and table 5.2 lists the different version and build numbers for the software and packages used.

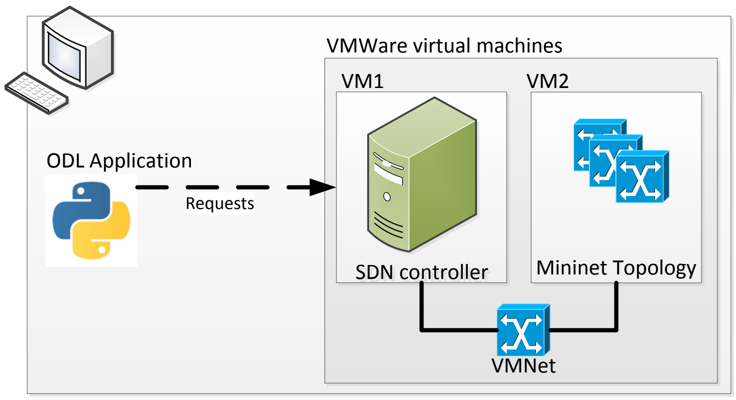


Figure 5.4: Test Bed

Item	Version
Open vSwitch in Mininet	2.0.0
Mininet	2.1.0
OpenDaylight Controller	0.1 SCM number 57f507d105b1daa9aa9663ca5ec6d258251fca2e
OS VM1	Ubuntu 13.04
OS VM2	Ubuntu 14.04
Python	2.7.6
Virtualization env.	VMWare Fusion Professional 5.0.4 (1435862)
Client OS	Mac OS X 10.9.2

Table 5.2: Software Versions used in Test Bed

5.3.1 Network Emulation in Mininet

Mininet[66, 67] were chosen early on as the preferred platform for emulating a network. Topologies with a sufficient number of nodes can easily be created through scripting (see annex B for an example script). Mininet is prominently used in many areas of network research and development and is easy to set up and understand[6]. Following is a short overview of the most important commands used in the test bed implementation.

To run a Mininet topology you have to use the command:

```
$ sudo mn
```

Running the basic *mn* command will create a default topology with two hosts connected to a single switch, using the default Mininet SDN controller. For more advanced topologies you would either have to use the built-in options for the *mn* command, or write a script. An example command using the built-in options could for instance be:

```
$ sudo mn -controller=remote,ip=192.168.231.246 -topo tree,3
-switch ovsk,protocols=OpenFlow13
```

Which will start a tree topology with three levels, connected to a remote controller and using the OVS kernel with OF version 1.3 support[7].

When using a script to build your Mininet topology, and you're using a remote controller you have to add this to the script file. For instance:

```
c0 = net.addController( 'c0', ip='127.0.0.1', port=6633)
```

This will connect the nodes to a remote controller in the same way as the -controller=remote,ip<ip-address> option.

The hosts created in Mininet runs in their own network namespaces in Linux, and the switches are set up as instances of OVS. This implies that each host emulated by Mininet can run all programs that are installed on the host client. The individual hosts can be given commands directly from the Mininet console:

```
mininet> h1 ping h2
```

[6]For more information about the use of Mininet, please refer to the the Mininet tutorial: http://mininet.org/walkthrough/ and chapter 5.2

[7]See chapter 6.3 for additional details

Which will run the *ping* command on host h1 towards h2. Another way is to open an Xterm terminal window for each of the hosts you want to work on. E.g:

```
mininet> xterm h1
```

5.3.2 SDN Controller

ODL were chosen as the preferred SDN controller for use in the test bed implementation. The controller were installed in a separate Linux Virtual Machine (VM) installation to not interfere with the network emulation in any way.

The details on how to install the ODL controller can be found on the wiki pages of the ODL project[8]. This was the same procedure as were used for installing the controller in the test bed.

To run and use the ODL controller with the implementation you have to add an option to start the controller with OF 1.3 support:

```
$ ./run.sh -of13
```

This starts OF 1.3 plugin module in the controller, and gives the possibility to use all functions in OF specification if the network devices support it. For the test bed we had to stop a module in the controller that provides simple switch forwarding (Learning switch module). This is done through a few commands in the OSGI command line interface

```
osgi> ss simple
osgi> 112     ACTIVE      org.opendaylight.controller
   .samples.simpleforwarding_0.4.1
osgi> stop 112
```

This will stop the simple forwarding bundle from interfering with the operation of policy application.

5.3.3 Policy Application

The following is a description of the policy application (implementation of the actual policy rule) developed for use in the test bed, depicted as *ODL Application* in figure 5.4.

The application provides a very simple command line based User Interface (UI). It can be used to manually add flows, add a SPF flow, look at the installed flows and delete installed

[8]https://wiki.opendaylight.org/view/OpenDaylight_Controller:Installation

flows. This is basic functionality that ODL controller doesn't have ”out of the box” support for when using the OF 1.3 plugin.

To test the designed policy implementation from chapter 4.2, Re-assign a tunnel, the application were developed to be able to move a tunnel. This is an implementation where a flow path between two hosts are created. To keep it simple, and to show that moving a tunnel with ease is achievable, the tunnel is no more than a defined path for flows between to IPv4 hosts. We believe that the same principle could also be used for traditional tunneling methods, e.g. IP-in-IP encapsulation, or with an attached label in the header such as in MPLS/MPLS-TE. In the application, the tunnel is not named as *TunnelX*, but rather a logical *srcIP, destIP* tuple. When the operator wants to move a tunnel, he will answer the following in the UI:

```
Welcome, what would you like to do? Type in number:
1. Add Flow
2. Look at flows
3. Delete flows
4. Move a flow
> 4
You want to move a flow
Between which hosts do you want to move the tunnel?
Source host >10.0.0.1
Destination host >10.0.0.2
Choose node to exclude from SPF calculation:
>openflow:2
```

When the input is added (marked in green) the program will first try to calculate a new path through the network, not using the excluded node. This is the same way as in MPLS-TE (ref. chapter 4.2.2), where you can explicitly state a node that should be excluded from the path calculation. It will then search through all nodes and look for active flow rules. It will then look at the active rules, and see if any of them have a srcIP or destIP matching the input in their match fields. If so, the rule will be deleted. After all old rules are deleted, new flow rules will be sent to the devices along the calculated path. A message will be prompted after flow rule installation.

```
The new path is: [u'openflow:1', u'openflow:3', u'openflow:4',
u'openflow:5']
```

5.4 Testing and Validation

This section describes the testing and validation phase of this thesis, were the object was to ensure that the test bed operates as expected.

The network topology used in the testing and validation phase were implemented using the script in annex B, and can be seen in figure 4.2. Please note that the logical names for the nodes as provided by the controller is *openflow:1-openflow:5* instead of *R1-R5*. Two hosts are connected to the network, *h1* connected to *openflow:1*, and *h2* connected to *openflow:5*.

To test the implementation a few different methods were used. As this is a small and controlled test environment, the flow rule tables where inspected manually before and after the tunnel move. This showed that the correct flow rules had been changed, and that all old flow rules had been deleted. The tunnel path were during the move changed from [openflow:1, openflow:2, openflow:5]$_{\sim 7\text{ Gbps}}$ to [openflow:1, openflow:3, openflow:4, openflow:5]$_{\sim 7\text{ Gbps}}$. Please note that the bandwidth on all links are the same in the beginning of the test phase. The bandwidth of ~7 Gbps were the maximum bandwidth possible in Mininet running on the provided hardware[9]. The results can be seen in listings C.1-C.4 in annex C.

The XML bodies in these listings shows clearly that the flow rules on node *openflow:2* and *openflow:3* have been updated during the tunnel move. When the flow rules have been moved from one node (i.e. deleted), the XML response will be `No data exists`. Please note that the listings only shows the stored flow rules in flow table 0 of each node. The ODL application only operates on flow table 0 of the network nodes. There are however 256 tables in total on each OVS instance that could have been used for creating flow rules in a pipe lined fashion (ref. chapter 2.1.1). All other tables (in the range 1-255) were empty during the testing and validation phase. This is ensured by the ODL application, as it inspects all active tables on all nodes.

To look at the performance of the network during a move, two methods where used. A *ping* command were issued between the two hosts, while two tunnel moves where executed. It is clear from the print out that the tunnel move caused longer latencies and packet loss. The packet loss is due to the flow rules being deleted before new ones are installed. This causes the switch to *drop* packets because no matches are found in the flow tables. This is the expected behavior. The following shows part of the ping command print out. Instances of increased latency and packet loss are marked with red, and *[...]* is used to denote omitted parts of the print out:

```
64 bytes from 10.0.0.2: icmp_seq=20 ttl=64 time=0.055 ms
64 bytes from 10.0.0.2: icmp_seq=21 ttl=64 time=0.406 ms
64 bytes from 10.0.0.2: icmp_seq=22 ttl=64 time=0.043 ms
```

[9]2012 Macbook Pro, 2.6 Ghz Intel Core i7, 16GB 1600MHz DDR3 Memory

```
[...]
64 bytes from 10.0.0.2: icmp_seq=37 ttl=64 time=0.094 ms
64 bytes from 10.0.0.2: icmp_seq=39 ttl=64 time=0.260 ms
64 bytes from 10.0.0.2: icmp_seq=40 ttl=64 time=0.045 ms
```

Secondly, an *iperf* test[10] were run between the two hosts to look at bandwidth degradation during the tunnel move. TCP was used as transmission protocol. The TCP variant used in the test bed were CUBIC [70]. A simple test showed a dip in measured bandwidth during the tunnel move because of packet loss. A graph has been made to show this:

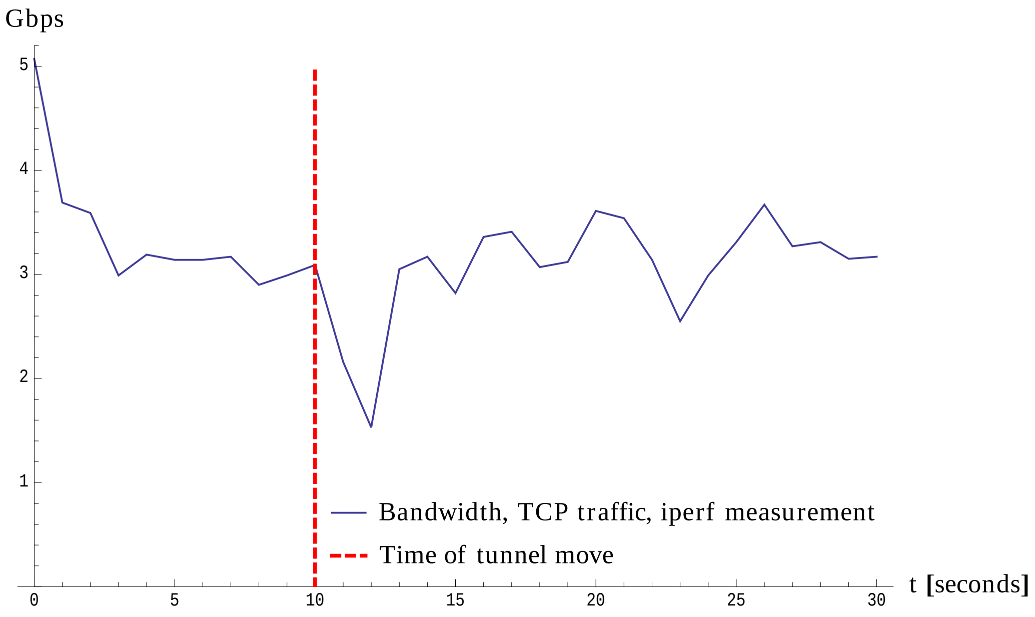


Figure 5.5: Measured Bandwidth During a Tunnel Move (TCP traffic)

In figure 5.5 the move were done right after the 10 second mark (measurement points is taken in 1 (one) second intervals). The expected result from packet drops is here seen, as the bandwidth is reduced with approx. 50%. This is in correspondence with what you should expect when using the TCP protocol [71]. However, the value of this graph for validation is rather small. In essence, it only shows that the *slow start* mechanism of TCP works. It also shows very large bandwidth fluctuations. A better test would therefore be to move the tunnel to a path with less bandwidth. The same test (using iperf) were conducted on a network where the tunnel were moved from $[R1,R2,R5]_{1000Mbps}$ to $[R1,R3,R4,R5]_{100Mbps}$, and then back to $[R1,R2,R5]_{1000Mbps}$. The changes in the network were done through changing

[10]Command: `iperf -c 10.0.0.2 -P 1 -i 1 -p 5001 -w 56K -M 1K -l 2M -f k -t 120`

the following lines of code in the Mininet start-up script (ref. annex B):

```
linkoptsA = dict(bw=1000, delay='0ms', loss=0)#Bandwith for path [R1,R2,
    R5] and host to switch
linkoptsB = dict(bw=100, delay='0ms', loss=0)#Bandwidth for secondary
    path [R1,R3,R4,R5]
print "*** Add links between switches ***"
net.addLink(switches[0],switches[1],**linkoptsA)
net.addLink(switches[0],switches[2],**linkoptsB)
net.addLink(switches[2],switches[3],**linkoptsB)
net.addLink(switches[3],switches[4],**linkoptsB)
net.addLink(switches[1],switches[4],**linkoptsA)
print "*** Add links to hosts ***"
net.addLink(switches[0],h1,**linkoptsA)
net.addLink(switches[4],h2,**linkoptsA)
```

Listing 5.6: Changes in Script for Different Bandwidth

A rerun of the iperf test were then conducted, with the following results:

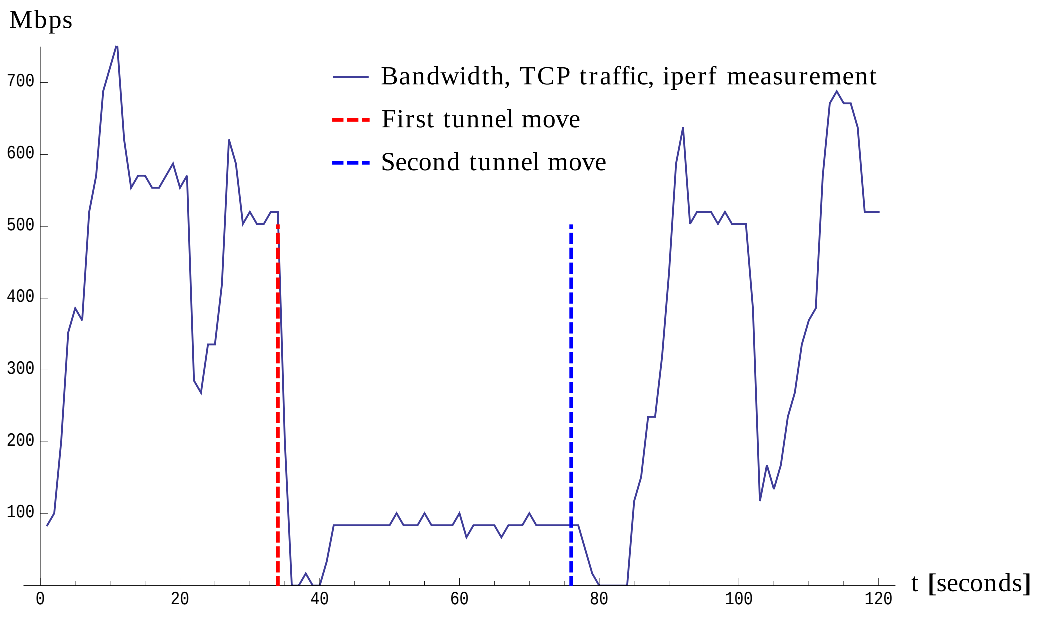


Figure 5.6: Measured Bandwidth During a Tunnel Move (TCP traffic). Different BW on Paths

In this figure (ref. figure 5.6) it is clear that the bandwidth changes when the tunnel is moved between the two paths. This validates that the tunnel is moved back and forth successfully. However, the graph shows massive bandwidth fluctuations when the tunnel is on path $[R1,R2,R5]_{1000Mbps}$. The reason for this is unknown. No other traffic than iperf and OF packets were present on the Mininet network during the test.

Chapter 6

Experiences From Implementation

This chapter is a description of our experiences obtained during the course of the implementation work. It covers learning points, challenges and functionality missing in the implementation in its current form.

The following table is recap of table 4.1 from chapter 4.3. Here we take a look at the stated ambitions for the implementation, and regard if they are solved or unsolved.

Part	**Ambition solved/unsolved**
SDN controller	OpenDaylight were chosen as controller. Provides an ample degree of functionality. Very complex with a large framework. Difficult framework for development. A simpler controller with less functionality and easier development, such as Pox [53], might have been more suited for this work. Especially regarding time constraints and size of a master thesis.
Network Emulation	Implemented in Mininet. Provides a flexible framework for testing network topologies. Can be expanded with Maxinet [68]. Does not have support for OF version 1.3 as of yet.
Policy application	We were not able to create an application at the control plane. Application created at the application plane. Uses northbound REST API to communicate with controller. Written in Python. Will not install flow rules reactively. No support for receiving policy updates. Policies must be written manually.
Reactive rule installation	Not solved. Not able to receive *packet_in* messages. Depends on control plane implementation.
Proactive rule installation	Solved, but in a manual fashion except for SPF and movement of tunnels.
Dynamic Enforcement	Not solved. Will have to pull information from the controller to see topology changes. Depends on control plane implementation.

Table 6.1: Results - Ambitions for the Implementation

From the table it's clear to see that several ambitions were not solved. The most important points are described further in section 6.1, 6.2 and 6.3.

6.1 Placement of the Policy Application

When the implementation work commenced, the initial thought was to create a new software module in the controllers OSGi framework using Maven [72]. This would mean the creation of a program written in Java that could interact with direct method calls against the other modules of the controller. It would also make it possible to create listeners that could pick up on changes in the topology, and to receive new packets entering the network directly (i.e. encapsulated packets sent between the controller and network device using the packet_in/packet_out function in the OF protocol). All through using the MD-SAL shared message bus in ODL. This was considered the most powerful and best solution to implement a policy application. However, the framework were unknown to the authors and it proved very difficult to set up a working development environment. The framework relies on development using the Eclipse code editor with a number of plugins. Also, special XML files have to be written so that new software bundles can be put into the OSGi framework. Considerable time were spent on trying to set up and learn the development environment, but it proved unsuccessful. It was therefore decided to create the ODL application in the application plane instead, using the Python programming language.

Placing the application in the application plane, rather than the control plane, caused a situation were several ambitions were rendered impossible to solve. Specifically *reactive rule installation* and *dynamic enforcement*. In a broad sense the latter could have been solved at the application plane, but only in very a sluggish and sub optimal way. An application placed in the application plane can not respond to dynamic topology changes in real time. It would have to request for information from the controller at regular intervals, check for changes, and then respond to these changes accordingly. It would *not* be possible to implement reactive rule installation as it's not possible to retrieve *packet_in* messages sent to the controller over the north bound REST API. With a control plane implementation, dynamic enforcement and reactive rule installation would have been possible.

6.2 OpenDaylight Complexity

Apart from requiring a complex development environment, the ODL controller is very complex in itself. It's built with a large number of modules that provides high degree of functionality and possibilities. It's clear that the controller should be able to operate in large number of scenarios, including enterprise data center and core networks. This shows in the large code base. A large amount of documentation has been written for the controller, primarily at [73], but it's difficult to navigate through it. This caused a situation were approx. a week amount of work were done on the wrong API. Namely on the Application-Driven Service Abstraction Layer (AD-SAL) instead of MD-SAL, which has more possibilities.

Another experienced challenge with using the OpenDaylight controller is that it's very new. Its first official release were at February 5th 2014 [60]; just at the start of the thesis work. This meant that there were very little examples or tutorials to look at, as very few developers had started developing applications for it.

6.3 Problem with Mininet 2.1

The tutorials found online about Mininet 2.1 and OVS version 2.0 will tell you that it has support for OF version 1.3. This is not the whole truth, and were discovered late in the work on the test bed implementation. OVS version 2.0 which is implemented in Mininet does not have support for group tables, which is a part of the OF 1.3 specification. This will not be implemented until the next revision of OVS[1]. This took quite some time to realize, as an initial thought were to use of the group tables, described in the OF 1.3 specification, in a policy covering a dependability scheme (path protection).

[1]Details can be found here: https://www.mail-archive.com/discuss@openvswitch.org/msg07882.html

Chapter 7 Conclusions

In this thesis we have shown that Software-Defined Networking can be used for policy enforcement in a federated military network. This has also been proven through a minimal policy rule implementation in a test bed, where a tunnel is moved between two paths of a network. The design phase of the implementation has also shown that tunnels in a network can be moved with more ease using SDN compared to MPLS-TE.

We have also found that dynamic enforcement of policies, and reactive flow rule installation, is possible if implemented in the control plane of the OpenDaylight controller. It is also described how this is not possible when implemented at the application plane.

The thesis also describes challenges connected with federated military networks, and possible solutions for controller placement in such a network. Following this, it has been found that more research is needed towards implementation of an east/west communication protocol between controllers of a federated network.

SDN is said to be a paradigm shift in networking, possibly solving many challenges that we find in traditional networking today. *Flexibility to create new solutions* is an often heard selling points for SDN. However, while SDN brings with it the possibility to create new solutions in the network through software programming, it also brings with it complexity. Complexity in the sense that it's not straight forward, or easy, to develop a high performance SDN solution. This thesis, through the implementation work, has shown this. The research area is still quite new, and few commercial off-the-shelf solutions are yet available. We assume that the research will continue in this field, and that in time, more solutions will be developed. This will show clearer how SDN best can be utilized in federated military networks.

7.1 Future Work

List of possible topics that could be studied further:

- Implement a federated network with east/west communication between controllers.
- Implement a policy engine as a software bundle in the OSGi framework of the OpenDaylight controller, to support real time dynamic capabilities.

References

[1] B. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti, “A survey of software-defined networking: Past, present, and future of programmable networks,” *Communications Surveys Tutorials, IEEE*, vol. PP, no. 99, pp. 1–18, 2014.

[2] J. Eckert, “Answering Questions on the Future Mission Network.” http://www.act.nato.int/article-2013-1-16, retrieved okt 20. 2013, 2013.

[3] G. Hallingstad and S. Oudkerk, “Protected core networking: an architectural approach to secure and flexible communications,” *Communications Magazine, IEEE*, vol. 46, no. 11, pp. 35–41, 2008.

[4] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, “B4: Experience with a globally-deployed software defined wan,” in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM ’13, (New York, NY, USA), pp. 3–14, ACM, 2013.

[5] Uninett, “https://www.uninett.no/en/traffic, retrieved may 12. 2014.”

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: enabling innovation in campus networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[7] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, “RFC 6241: Network Configuration Protocol (NETCONF),” 2011.

[8] IEEE, “802.1AB Station and Media Access Control Connectivity Discovery,” 2009.

[9] T. D. Nadeau and K. Gray, *SDN: Software Defined Networks.* O’Reilly, 2013.

[10] A. Atlas, J. Halpern, S. Hares, D. Ward, and T. D. Nadeau, “An Architecture for the Interface to the Routing System.” Informational Internet-Draft, 2013.

[11] A. T. Campbell and I. Katzela, “Open Signaling for ATM, Internet and Mobile Networks (Opensig’98),” 1999.

[12] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, “A survey of active network research,” *Communications Magazine, IEEE*, vol. 35, no. 1, pp. 80–86, 1997.

[13] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 81–94, Oct. 2007.

[14] J. E. Van Der Merwe and I. M. Leslie, "Switchlets and dynamic virtual atm networks," in *Proc Integrated Network Management V*, pp. 355–368, Chapman and Hall, 1997.

[15] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *SIGCOMM Comput. Commun. Rev.*, vol. 35, pp. 41–54, Oct. 2005.

[16] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 1–12, Aug. 2007.

[17] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)." RFC, January 2006.

[18] W. Stallings, "Software-Defined Networks and OpenFlow," *The Internet Protocol Journal*, vol. 16, pp. 2–14, March 2013.

[19] ONF, "Software-defined networking: The new norm for networks," white paper, Open Networking Foundation, April 2012.

[20] ONF, *OpenFlow Switch Specification Version 1.3.3 (Protocol Versoin 0x04).* Open Networking Foundation, September 2013.

[21] OpenDaylight Foundation, "https://www.opendaylight.org/, retrieved march 23. 2014."

[22] S. Raza and D. Lenrow, "ONF North Bound Interface Working Group (NBI-WG) Charter," tech. rep., ONF, 2013.

[23] Cisco, "Technical Overview of onePK." http://developer.cisco.com/web/onepk-developer/technical-overview, retrieved 24. nov 2013.

[24] B. Pfaff and B. Davie, "The Open vSwitch Database Management Protocol," October 2013.

[25] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *In Proc. of PODC*, pp. 398–407, ACM Press, 2007.

[26] E. Rosen, A. Viswanathan, and R. Callon, "RFC 3031: Multiprotocol Label Switching Architecture," 2001.

[27] L. Ghein, *MPLS Fundamentals - A Comprehensive Introduction to MPLS Theory and Practice.* Cisco Press, 2007.

[28] J. Moy, "RFC 2328: OSPF Version 2," April 1998.

[29] R. Coltun, D. Ferguson, J. Moy, and A. Lindem, "RFC 5340: OSPF for IPv6," July 2008.

[30] G. Malkin, "RFC 2453: RIP version 2," November 1998.

[31] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica, "Pathlet routing," *SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 111–122, Aug. 2009.

[32] G. Andersson, L. & Swallow, "RFC 3468: The Multiprotocol Label Switching (MPLS) Working Group decision on MPLS signaling protocl," 2003.

[33] Cisco Systems, *MPLS Traffic Engineering (TE): Path Protection.* Cisco, Cisco Systems, Inc., 170 West Tasman Drive, San Jose, CA 95134-1706 USA, July 2011.

[34] G. Waters, J. Wheeler, A. Westerinen, L. Rafalow, and R. Moore, *Policy Framework Architecture.* IETF, internet-draft ed., February 1999.

[35] K. Hyojoon and N. Feamster, "Improving network management with software defined networking," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 114–119, 2013.

[36] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, (New York, NY, USA), pp. 43–48, ACM, 2012.

[37] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, (New York, NY, USA), pp. 279–291, ACM, 2011.

[38] D. Coppieters and J. van Geest, "Future Mission Training," Tech. Rep. STO-MP-MSG-094, NATO Communications and Information Agency, Unknown year.

[39] T. G. RTG-032/IST-069, "Requirements for a Protected Core Networking (PCN) Interoperability Specification (ISpec)," Tech. Rep. TR-IST-069, NATO, July 2012.

[40] C. Semaria and J. W. Stewart, "Supporting Differentiated Service Classes in Large IP Networks," white paper, Juniper Networks, 2001.

[41] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "RFC 2475: An Architecture for Differentiated Services," 1998.

[42] E. Sorensen, "Evaluating Software Defined Networking for Use in Military Networks." Unpublished Preliminary Project Report, November 2013.

[43] W. Scull, "Introducing IBM Software Defined Network for Virtual Environments (SDN VE) VMware Edition," tech. rep., IBM Systems and Technology Group, 2014.

[44] H. Yin, H. Xie, T. Tsou, D. Lopez, P. Aranda, and R. Sidi, "SDNi: A Message Exchange Protocol for Software Defined Networks (SDNS) across Multiple Domains." Internet-Draft, June 2012.

[45] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. 254–265, Aug. 2011.

[46] G. Fayolle, A. de la Fortelle, J.-M. Lasgouttes, L. Massoulie, and J. Roberts, "Best-effort networks: modeling and performance analysis via large networks asymptotics," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 709–716 vol.2, 2001.

[47] H. Vo, O. Lysne, and A. Kvalbein, "Permutation routing for increased robustness in ip networks," in *NETWORKING 2012* (R. Bestak, L. Kencl, L. Li, J. Widmer, and H. Yin, eds.), vol. 7289 of *Lecture Notes in Computer Science*, pp. 217–231, Springer Berlin Heidelberg, 2012.

[48] B. E. Helvik, *Dependable Computing Systems and Communication Networks - Design and Evaluation.* Department of Telematics, NTNU, 2009.

[49] Linux Foundation, "https://git.opendaylight.org/gerrit/p/controller.git, retrieved march 23. 2014."

[50] T. Fielding, *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

[51] NOXRepo.org, "http://www.noxrepo.org/nox/about-nox/, retrieved march 23. 2014."

[52] NOXRepo.org, "http://noxrepo.org/git/nox, retrieved march 23. 2014."

[53] NOXRepo.org, "http://www.noxrepo.org/pox/about-pox/, retrieved march 23. 2014."

[54] NOXRepo.org, "https://github.com/noxrepo/pox, retrieved march 23. 2014."

[55] D. Erickson, "The Beacon OpenFlow Controller," in *HotSDN*, ACM, 2013.

[56] Stanford University, "https://openflow.stanford.edu/display/Beacon/Home, retrieved march 23. 2014."

[57] Project Floodlight, "http://www.projectfloodlight.org, retrieved march 23. 2014."

[58] Project Floodlight, "https://github.com/floodlight/floodlight, retrieved march 23. 2014."

[59] Nippon Telegraph and Telephone Company, "http://osrg.github.io/ryu/, retrieved march 23. 2014."

[60] Tech Target, "OpenDaylight Hydrogen release and what you can do with it, retrieved june 11. 2014."

[61] OpenDaylight Foundation, "Technical Overview." http://www.opendaylight.org/project/technical-overview, retrieved nov 24. 2013.

[62] OSGi Alliance, "http://www.osgi.org/Main/HomePage, retrieved junw 2. 2014."

[63] Linux Foundation, "https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:MD-SAL_App_Tutorial, retrieved june 2. 2014."

[64] Linux Foundation, "https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Model_Reference, retrieved march 25. 2014."

[65] CPqD/ofsoftswitch13 , “https://github.com/CPqD/ofsoftswitch13, retrieved 23. april 2014.”

[66] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, (New York, NY, USA), pp. 19:1–19:6, ACM, 2010.

[67] Mininet.org, “Mininet - an instant virtual network on your laptop (or other pc) http://www.mininet.org, retrieved april 24. 2014.”

[68] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, and H. Zahraee, “MaxiNet: Distributed Emulation of Software-Defined Networks,” in *IFIP Networking 2014 Conference (Networking 2014)*, IFIP, 2014.

[69] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, and H. Zahraee, “Maxinet: Distributed software defined network emulation http://www.cs.uni-paderborn.de/fachgebiete/fachgebiet-rechnernetze/people/philip-wette-msc/maxinet.html, retrieved may 14. 2014.”

[70] S. Ha, I. Rhee, and L. Xu, “Cubic: A new tcp-friendly high-speed tcp variant,” *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 64–74, July 2008.

[71] L. Parziale, D. Britt, C. Davis, J. Forrster, W. Liu, C. Matthews, and N. Rosseblot, *TCP/IP Tutorial and Technical Overview.* IBM Redbooks, 2006.

[72] Apache Maven Project, “https://maven.apache.org/what-is-maven.html, retrieved june 6. 2014.”

[73] Linux Foundation, “https://wiki.opendaylight.org/, retrieved june 11. 2014.”

Appendix A
Test Bed Implementation Code

This appendix includes the following files from the test best implementation.

- **odl-application.py**: The main part of the ODL application used to interact with the controller.
- **frontend.py**: The main part of the console based user interface
- **restconf.py**: Functions used to communicate with the controller using REST.
- **actions.txt**: Text file containing possible flow rule actions. Used for reference in the application.
- **matches.txt**: Text file containing possible flow rule matches. Used for reference in the application.

odl−application.py

```
#External libraries
import sys
import json
import networkx as nx
from networkx.readwrite import json_graph
import httplib2
from xml.dom import minidom
from lxml import etree
#Own libraries
import restconf
import frontend

#Base URLs for Config and operational
baseUrl = 'http://192.168.231.250:8080'
confUrl = baseUrl + '/restconf/config/' #Contains data inserted via
    controller
operUrl = baseUrl + '/restconf/operational' # Contains other data
findTopology = operUrl + '/network-topology:network-topology/topology/flow
    :1/'

h = httplib2.Http(".cache")
#Username and password for controller
h.add_credentials('admin', 'admin')
#Counter used when creating flows
flowIdCounter = int(100)
#Create a list with active hosts in the network
hosts = restconf.get_active_hosts()

#Function to find topology parameters
def get_topology(xml):
    topology = json.loads(xml)
    print topology
    nodes = topology['topology'][0]['node']
    links = topology['topology'][0]['link']
    return topology
#Function to find the shortest path between two nodes in the network
def get_sp(topology, src, dst):
    graph = nx.Graph()
    nodes = topology['topology'][0]['node']
    links = topology['topology'][0]['link']
    for node in nodes:
        graph.add_node(node['node-id'])
    for link in links:
        e = (link['source']['source-node'], link['destination']['dest-node'
            ])
        graph.add_edge(*e)
    #Using the networkx library to calculate shortest path
    sp = nx.shortest_path(graph, src, dst)
    return sp
#Function to find which node a host is connected to
def host_switch(hosts, IP):
    for host in hosts:
        if host['networkAddress'] == IP:
            switch = host['nodeId']
    return switch#If an error is thrown here, you have to do a 'pingall' on
         mininet. This is caused by a bug in ODL
#Function to find which port of node a host is connected to
def host_port(hosts, IP):
    for host in hosts:
        if host['networkAddress'] == IP:
            switchport = host['nodeConnectorId']
    return switchport#If an error is thrown here, you have to do a 'pingall
        ' on mininet. This is caused by a bug in ODL
#Function to fint port numbers for links between nodes
def find_ports(xml, headNode, tailNode):
    links = xml['topology'][0]['link']
    for link in links:
        if link['source']['source-node'] == headNode and link['destination'
            ]['dest-node'] == tailNode:
            portId = link['source']['source-tp']
            return portId
    return None
#Function to add shortest path flows
def add_sp_flows(shortest_path, srcIP, dstIP):
    flowId = flowIdCounter
    hardTimeOut, idleTimeOut = '0','0'
    #Create flow rules to directly connected hosts from headnode and
        tailnode
    #HEAD
    flowName = shortest_path[0] + 'to' + srcIP
    outPutPort = host_port(hosts, srcIP)
    hostURL = confUrl+'opendaylight-inventory:nodes/node/'+shortest_path
        [0]+'/table/0/flow/'+str(flowId)
    #Because of a bug in ODL when updating flows we have to delete
    #any flows with the same flow id:
    restconf.delete(hostURL)
    flow = flow_rule_base(flowName, '0', str(flowId), hardTimeOut,
        idleTimeOut)
    flow = add_flow_action_sp(flow, outPutPort)
    flow = add_flow_match_sp(flow, srcIP)
    XMLstring = etree.tostring(flow,pretty_print=True,xml_declaration=True,
         encoding="utf-8", standalone=False )
    restconf.put(hostURL, XMLstring)
    #TAIL
    flowId = flowId + 1
    flowName = shortest_path[-1] + 'to' + dstIP
    outPutPort = host_port(hosts, dstIP)
    hostURL = confUrl+'opendaylight-inventory:nodes/node/'+shortest_path
        [-1]+'/table/0/flow/'+str(flowId)
    restconf.delete(hostURL)
    flow = flow_rule_base(flowName, '0', str(flowId), hardTimeOut,
        idleTimeOut)
    flow = add_flow_action_sp(flow, outPutPort)
    flow = add_flow_match_sp(flow, dstIP)
    XMLstring = etree.tostring(flow,pretty_print=True,xml_declaration=True,
         encoding="utf-8", standalone=False )
    restconf.put(hostURL, XMLstring)
    #For loop to create flow rules between all OF nodes end to end
    for i in range(len(shortest_path)-1):
        headNode = shortest_path[i]
        tailNode = shortest_path[i+1]
        flowId = flowId + 1
        #Forward Flow
        flowName = headNode + 'to' + tailNode + 'IPto' + dstIP
        outPutPort = find_ports(get_topology(restconf.get(findTopology)),
            shortest_path[i], shortest_path[i+1])
        forwardURL = confUrl+'opendaylight-inventory:nodes/node/'+
            shortest_path[i]+'/table/0/flow/'+str(flowId)
        restconf.delete(forwardURL)
        flow = flow_rule_base(flowName, '0', str(flowId), hardTimeOut,
            idleTimeOut)
        flow = add_flow_action_sp(flow, outPutPort)
        flow = add_flow_match_sp(flow, dstIP)
        forwardXMLstring = etree.tostring(flow,pretty_print=True,
            xml_declaration=True, encoding="utf-8", standalone=False )
        restconf.put(forwardURL, forwardXMLstring)
        #Backward Flow
        flowName = tailNode + 'to' + headNode + 'IPto' + srcIP
        outPutPort = find_ports(get_topology(restconf.get(findTopology)),
            shortest_path[i+1], shortest_path[i])
        backwardURL = confUrl+'opendaylight-inventory:nodes/node/'+
            shortest_path[i+1]+'/table/0/flow/'+str(flowId)
        restconf.delete(backwardURL)
        flow = flow_rule_base(flowName, '0', str(flowId), hardTimeOut,
            idleTimeOut)
        flow = add_flow_action_sp(flow, outPutPort)
        flow = add_flow_match_sp(flow, srcIP)
        backwardXMLstring =  etree.tostring(flow,pretty_print=True,
            xml_declaration=True, encoding="utf-8", standalone=False )
        restconf.put(backwardURL, backwardXMLstring)
#Function to build the base of a flow rule
def flow_rule_base(flowName, tableId, flowId, hardTimeout, idleTimeout):
    flow = etree.Element("flow")
    flow.set('xmlns','urn:opendaylight:flow:inventory')
    strict = etree.SubElement(flow, "strict")
    strict.text = "false"
    flow_name = etree.SubElement(flow, "flow-name")
    flow_name.text = flowName
    id = etree.SubElement(flow, "id") #ID of the flow
    id.text = flowId
    table_id = etree.SubElement(flow, "table_id") #ID of the table on that
        switch
    table_id.text = tableId
    hard_timeout = etree.SubElement(flow, "hard-timeout")
    hard_timeout.text = hardTimeout
    idle_timeout = etree.SubElement(flow, "idle-timeout")
    idle_timeout.text = idleTimeout
    priority = etree.SubElement(flow, "priority")
    priority.text = "1"
    cookie = etree.SubElement(flow, "cookie")
    cookie.text = "0"
    barrier = etree.SubElement(flow, "barrier")
    barrier.text = "false"
    cookie_mask = etree.SubElement(flow, "cookie_mask")
    cookie_mask.text = "255"
    installHw = etree.SubElement(flow, "installHw")
    installHw.text = "True"

    #The Actions
    instructions = etree.SubElement(flow, "instructions")
    instruction = etree.SubElement(instructions, "instruction")
    order_instruct = etree.SubElement(instruction, "order")
    order_instruct.text = "0"
    apply_actions = etree.SubElement(instruction, "apply-actions")
    action = etree.SubElement(apply_actions, "action")
    order_action = etree.SubElement(action, "order")
    order_action.text = "0"

    #The Matches
    match = etree.SubElement(flow, "match")
    ethernet_match = etree.SubElement(match, "ethernet-match")
    ethernet_type = etree.SubElement(ethernet_match, "ethernet-type")
    type = etree.SubElement(ethernet_type, "type")
    type.text = "2048"
    return flow
#function to add a match field to a flow rule
def add_flow_match_sp(flow, destination):
    mat = flow.xpath('//match')[0]
    ipv4d = etree.SubElement(mat, 'ipv4-destination')
    ipv4d.text = destination
    return flow
#Function to add an action to a flow rule
def add_flow_action_sp(flow, port):
    action = flow.xpath('//action')[0]
    _act = etree.SubElement(action, 'output-action')
    onc = etree.SubElement(_act, 'output-node-connector')
    onc.text = port
    ml = etree.SubElement(_act, 'max-length')
    ml.text = '600'
    return flow
#Function to exclude node from path calculation
def exclude_switch_spf(nonSwitch, topology, src, dst):
    graph = nx.Graph()
    nodes = topology['topology'][0]['node']
    #nodes.remove(nonSwitch)
    links = topology['topology'][0]['link']
    for node in nodes:
        graph.add_node(node['node-id'])

    for link in links:
        e = (link['source']['source-node'], link['destination']['dest-node'
            ])
        graph.add_edge(*e)
    graph.remove_node(nonSwitch)
    sp = nx.shortest_path(graph, src, dst)
    return sp
#Function to delete old flow rules when moving a flow
def move_delete_old(srcIP, destIP):
    nodes = restconf.get_topology(restconf.get(findTopology))['topology'
        ][0]['node']
    for node in nodes:
        tables = restconf.get('http://192.168.231.250:8080/restconf/
            operational/opendaylight-inventory:nodes/node/'+node['node-id'
            ])
        flowTables = json.loads(tables)
        try:
            for table in flowTables['node'][0]['flow-node-inventory:table'
                ]:
                if table['opendaylight-flow-table-statistics:flow-table-
                    statistics']['opendaylight-flow-table-statistics:active
                    -flows'] != 0:
                    try:
                        flowRules = restconf.get(confUrl+'opendaylight-
                            inventory:nodes/node/'+node['node-id']+'/table/
                            '+str(table['flow-node-inventory:id']))
                        rules = json.loads(flowRules)
                        for rule in rules['flow-node-inventory:table'][0]['
                            flow-node-inventory:flow']:
                            if rule['flow-node-inventory:match']['flow-node
                                -inventory:ipv4-destination'] == srcIP:
                                tableID = str(table['flow-node-inventory:id
                                    '])
                                flowID = str(rule['flow-node-inventory:id'
                                    ])
                                url = confUrl+'opendaylight-inventory:nodes
                                    /node/'+node['node-id']+'/table/'+
                                    tableID+'/flow/'+flowID
                                restconf.delete(url)
                            elif rule['flow-node-inventory:match']['flow-
                                node-inventory:ipv4-destination'] == destIP
                                :
                                tableID = str(table['flow-node-inventory:id
                                    '])
                                flowID = str(rule['flow-node-inventory:id'
                                    ])
                                url = confUrl+'opendaylight-inventory:nodes
                                    /node/'+node['node-id']+'/table/'+
                                    tableID+'/flow/'+flowID
                                restconf.delete(url)
                except ValueError:
                    pass
        except KeyError:
            pass


#Main program
def program():
    answer = frontend.main_menu()
    if answer == 'addFlow':
        answer = frontend.show_act_mat()
        if answer == 'addFlow':
            _node, _tableId, _flowId, _flowName, _hardTimeOut, _idleTimeOut
                 = frontend.add_flow_gui()
            newFlow = flow_rule_base(_flowName, _tableId, _flowId,
                _hardTimeOut, _idleTimeOut)
            newFlow = frontend.add_actions(newFlow)
            newFlow = frontend.add_matches(newFlow)
            print etree.tostring(newFlow, pretty_print=True,xml_declaration
                =True, encoding="utf-8", standalone=False)
            print restconf.put(confUrl+'opendaylight-inventory:nodes/node/
                openflow:1/table/'+_tableId+'/flow/'+_flowId, etree.
                tostring(newFlow, xml_declaration=True, encoding="utf-8",
                standalone=False))
        elif answer == 'spfFlow':
            srcHost, destHost = frontend.get_ip_spf()
            shortest_path = get_sp(get_topology(restconf.get(findTopology))
                , host_switch(hosts, srcHost), host_switch(hosts, destHost)
                )
            print "The shortest path between host %s and %s follows the
                following path:\n" % (srcHost,destHost) +str(shortest_path)
            print "Would you like to add this flow? (y/n) "
            answer = frontend.yes_no()
            if answer == 'y':
                add_sp_flows(shortest_path, srcHost, destHost)
                print "Your shortest path flow is added to the switches"
            else:
                pass
        else:
            pass
    elif answer == 'lookFlows':
        print json.dumps(frontend.view_flows(), indent=2)
        print '\n'
        frontend.main_menu()
    elif answer == 'delFlow':
        frontend.del_flow()
    elif answer == 'moveFlow':
        nonSwitch, srcHost, destHost = frontend.move_flow()
        topo = get_topology(restconf.get(findTopology))
        newPath = exclude_switch_spf(nonSwitch, topo, host_switch(hosts,
            srcHost), host_switch(hosts, destHost))
        print 'The new path is: '+str(newPath)
        move_delete_old(srcHost,destHost)
        add_sp_flows(newPath, srcHost, destHost)
    else:
        pass
    program()

program()
```

frontend.py

```
import restconf
import json
from lxml import etree
#Base URLs for Config and operational
baseUrl = 'http://192.168.231.250:8080'
confUrl = baseUrl + '/restconf/config/'
operUrl = baseUrl + '/restconf/operational/'
findTopology = operUrl + '/network-topology:network-topology/topology/flow
    :1/'
actionsTxt = open('actions.txt', 'r')
matchesTxt = open('matches.txt', 'r')
#Function to view flows in the topology
def view_flows():
    print 'On which switch do you want to look at the flows?'
    print 'Type in the number of the switch (as listed):'
    nodes = restconf.get_topology(restconf.get(findTopology))['topology'
        ][0]['node']
    for node in nodes:
        print node['node-id']
    answer = raw_input('> ')
    print 'Type in the number of the table you would like to look at:'
    answer2 = raw_input('> ')
    content = restconf.get('http://192.168.231.250:8080/restconf/config/
        opendaylight-inventory:nodes/node/openflow:'+answer+'/table/'+
        answer2+'/')
    flows = json.loads(content)
    return flows['flow-node-inventory:table'][0]['flow-node-inventory:flow'
        ]
#User input yes or no
def yes_no():
    answer = raw_input(' >')
    return answer
#Function to delete a flow manually
def del_flow():
    print 'On which node do you want to delete a flow?'
    node = raw_input('> ')
    print 'In which table of node '+node+' do you want to delete a flow?'
    table = raw_input('> ')
    print 'What is the flow id for the flow you want to delete?'
    flowId = raw_input('> ')
    print 'Do you really want to delete flow '+flowId+' in table '+table+'
        on node '+node+' ? (y/n)'
    answer = raw_input('> ')
    if answer == 'y':
        url = confUrl+'opendaylight-inventory:nodes/node/openflow:'+node+'/
            table/'+table+'/flow/'+flowId
        print restconf.delete(url)
    elif answer == 'n':
        del_flow()
    else:
        print 'You answered gibberish! Try again'
        del_flow()
#User input for host source and destination addresses
def get_ip_spf():
    srcHost = raw_input('Type IP of Source host > ')
    destHost = raw_input('Type IP of destination host >')
    return srcHost, destHost


def show_act_mat():
    print '\nYou chose to add a flow. Would you like to see your possible
        match and action fields? Type in number:'
    print '1. Show actions'
    print '2. Show instructions'
    print '3. Show both'
    print '4. Add manual flow'
    print '5. Add SPF flow'
    answer = raw_input('> ')
    if answer == '1':
        print actionsTxt.read()
        show_act_mat()
    elif answer == '2':
        print matchesTxt.read()
        show_act_mat()
    elif answer == '3':
        print actionsTxt.read()
        print matchesTxt.read()
        show_act_mat()
    elif answer == '4':
        return 'addFlow'
    elif answer == '5':
        return 'spfFlow'
    else:
        print 'You answered gibberish! Try again'
        show_act_mat()
    return None
#User input for flow specifics
def add_flow_gui():
    print 'You chose to add a flow. Please answer these parameters'
    print 'First the RESTConf specific parameters. E.g: /opendaylight-
        inventory:nodes/node/openflow:1/table/0/flow/1'
    node = raw_input('Node? > ')
    table = raw_input('Table? > ')
    flowId = raw_input('Flow number? > ')
    print 'Then the flow specifics:'
    flowName = raw_input('FlowName? > ')
    hardTimeOut = raw_input('Hard Time Out? > ')
    idleTimeOut = raw_input('Idle Time Out? > ')
    return node, table, flowId, flowName, hardTimeOut, idleTimeOut


#User input for actions
def add_actions(xml):
    print 'You need to add some actions to your flow'
    i = int(input('How many actions do you need to add? > '))
    print 'Write in your actions. Remember that they are: '
    print actionsTxt.read()
    while (i > 0):
        j = str(i)
        act = raw_input('Action '+j+' > ')
        if act == 'output-action':
            print '    You need to add some subelements to that one:'
            print '    physical port #, ANY, LOCAL, TABLE, INPORT, NORMAL,
                FLOOD, ALL, CONTROLLER'
            output_node_connector = raw_input('    > ')
            print '    And max length:'
            max_length = raw_input('    > ')
            action = xml.xpath('//action')[0]
            _act = etree.SubElement(action, act)
            onc = etree.SubElement(_act, 'output-node-connector')
            onc.text = output_node_connector
            ml = etree.SubElement(_act, 'max-length')
            ml.text = max_length
        else:
            action = xml.xpath('//action')[0]
            etree.SubElement(action, act)
        i = i - 1
    return xml
#User input for matches
def add_matches(xml):
    mat = xml.xpath('//match')[0]
    print 'You need to add some matches to your flow'
    i = int(input('How many matches do you need to add? > '))
    print 'Write in your matches. Remember that they are: '
    print matchesTxt.read()
    while (i > 0):
        j = str(i)
        match = raw_input('Match '+j+' > ')
        if match == 'ethernet-match':
            print '    The default Ethernet type is 2048. Do you need to
                change this? (y/n)'
            answer = raw_input('    >')
            if answer == 'y':
                e_type = xml.xpath('//ethernet-type')[0]
            else:
                pass
            print '    You need to add some subelements to that one:'
            print '    Source address? (y/n)?'
            ethernet_match = xml.xpath('//ethernet-match')[0]
            answer = raw_input('    >')
            if answer == 'y':
                es = etree.SubElement(ethernet_match, 'ethernet-source')
                es_address = etree.SubElement(es, 'address')
                address = raw_input('    Address >')
                es_address.text = address
            else:
                pass
            print '    Destination address? (y/n)'
            answer == raw_input('    >')
            if answer == 'y':
                ed = etree.SubElement(ethernet_match, 'ethernet-destination
                    ')
                ed_address = etree.SubElement(ed, 'address')
                address = raw_input('    Address >')
                ed_address.text = address
            else:
                pass
        elif match == 'ipv4-destination':
            answer = raw_input('    Address >')
            ipv4d = etree.SubElement(mat, match)
            ipv4d.text = answer
        elif match == 'ipv4-source':
            answer = raw_input('    Address >')
            ipv4s = etree.SubElement(mat, match)
            ipv4s.text = answer
        elif match == 'tcp-source-port':
            answer = raw_input('    Address >')
            tcpsp = etree.SubElement(mat, match)
            tcpsp.text = answer
        elif match == 'tcp-destination-port':
            answer = raw_input('    Address >')
            tcpdp = etree.SubElement(mat, match)
            tcpdp.text = answer
        elif match == 'udp-source-port':
            answer = raw_input('    Address >')
            udpsp = etree.SubElement(mat, match)
            udpsp.text = answer
        elif match == 'udp-destination-port':
            answer = raw_input('    Address >')
            udpdp = etree.SubElement(mat, match)
            udpdp.text = answer
        elif match == 'vlan-match':
            answer = raw_input('    VLAN ID >')
            vlanm = etree.SubElement(mat, match)
            vlanid = etree.SubElement(match, 'vlan-id')
            vlanid_ = etree.SubElement(vlanid, 'vlan-id')
            vlanid_.text = answer
            vlanidpresent = etree.SubElement(_vlanid, 'true')
            answer = raw_input('    VLAN PCP >')
            vlanpcp = etree.SubElement(match, 'vlan-pcp')
            vlanpcp.text = answer
        elif match == 'tunnel':
            answer = raw_input('    Tunnel ID >')
            tunnel = etree.SubElement(mat, match)
            tunnelid = etree.SubElement(match, 'tunnel-id')
            tunnelid.text = answer
        else:
            pass
        i = i -1
    return xml
#User input used when moving a tunnel
def move_flow():
    print 'Between which hosts do you want to move the tunnel?'
    srcHost = raw_input('Source host >')
    destHost = raw_input('Destination host >')
    print 'Choose node to exclude from SPF calculation:'
    nonSwitch = raw_input(' >')
    return nonSwitch, srcHost, destHost

#Main meno for the UI
def main_menu():
    print "Welcome, what would you like to do? Type in number:"
    print "1. Add Flow"
    print "2. Look at flows"
    print "3. Delete flows"
    print "4. Move a flow"
    answer = raw_input('> ')
    if answer == '1':
        return 'addFlow'
    elif answer == '2':
        print 'You chose to look at flows'
        return 'lookFlows'
    elif answer == '3':
        print 'You want to delete a flow'
        return 'delFlow'
    elif answer == '4':
        print 'You want to move a flow'
        return 'moveFlow'
    else:
        print 'You answered gibberish! Try again'
        main_menu()
```

restconf.py

```
import sys
import json
import httplib2

#Base URLs for Config and operational
baseUrl = 'http://192.168.231.250:8080'
confUrl = baseUrl + '/restconf/config'
operUrl = baseUrl + '/restconf/operational'

#"Old" REST APIs that still are used
sdSalUrl = baseUrl + '/controller/nb/v2/'

#Specific REST URLs
findNodes = operUrl + '/opendaylight-inventory:nodes/'
findTopo = operUrl + '/network-topology:network-topology/'
findNodeConnector = operUrl + '/opendaylight-inventory:nodes/node/node-
    connector/'
findTopology = operUrl + '/network-topology:network-topology/topology/flow
    :1/'
findFlow = confUrl +'/opendaylight-inventory:nodes/node/openflow:1/table/0/
    '
findTopology = operUrl + '/network-topology:network-topology/topology/flow
    :1/'
h = httplib2.Http(".cache")
h.add_credentials('admin', 'admin')

#GET function. Retrieve information
def get(url):
    resp, xml = h.request(
        url,
        method = "GET",
        headers = {'Content-Type' : 'application/xml'}
        )
    return xml
#Put function.
def put(url, body):
    resp, content = h.request(
        url,
        method = "PUT",
        body = body,
        headers = {'Content-Type' : 'application/xml', 'Accept':'
            application/xml'}
        )
    return resp, content
#DELETE function
def delete(url):
    resp, content = h.request(
        url,
        method = "DELETE"
        )
    return resp
```

```
#Find active hosts
def get_active_hosts():
    resp, content = h.request(sdSalUrl + 'hosttracker/default/hosts/active/
        ', "GET")
    hostConfig = json.loads(content)
    hosts = hostConfig['hostConfig']
    return hosts
#Find topology
def get_topology(xml):
    topology = json.loads(xml)
    nodes = topology['topology'][0]['node']
    links = topology['topology'][0]['link']
    return topology
```

actions.txt

```
###### Possible actions ######
- dec-nw-ttl
- dec-mpls-ttl
- output-action
    * output-node-connector (physical port #, ANY, LOCAL, TABLE, INPORT,
        NORMAL, FLOOD, ALL, CONTROLLER)
    * max-length
- flood-all-action
- drop-action

https://jenkins.opendaylight.org/openflowjava/job/openflowjava-verify/ws/
    openflow-protocol-api/target/site/restconf/openflow-action.html#output
##############################
```

matches.txt

```
###### Possible Matches ######
- inport (<in-port>0</in-port>)
- ethernet-match
    * ethernet-type
        - type
    * ethernet-source
        - address (<address>00:00:00:00:00:01</address>)
    * ethernet-destination
        - address
- ipv4-destination (<ipv4-destination>10.0.1.1/24</ipv4-destination>)
- ipv4-source
- ipv6-destination
- ipv6-source
- ipv6-label
    * ipv6-flabel
- ipv6-ext-header
    * ipv6-exthdr
- ip-match
    * ip-protocol (<ip-protocol>56</ip-protocol>)
    * ip-dscp (<ip-dscp>15</ip-dscp>)
    * ip-ecn (<ip-ecn>1</ip-ecn>)
- icmpv4-match
    * icmpv4-type
    * icmpv4-code
- icmpv6-match
    * icmpv6-type
    * icmpv6-code
- tcp-source-port
- tcp-destination-port
- udp-source-port
- udp-destination-port
- arp-op
- arp-source-transport-address
- arp-target-transport-address
- arp-source-hardware-address
    * address
- arp-taget-hardware-address
    * address
- vlan-match
    * vlan-id
        - vlan-id
        - vlan-id-present (<vlan-id-present>true</vlan-id-present>)
    * vlan-pcp
- protocol-match-fields
    * mpls-label
    * mpls-tc
    * mpls-bos
- metadata
    * metadata (<metadata>12345</metadata>)
    * metadata-mask (<metadata-mask>//FF</metadata-mask>)
```

```
- tunnel
    * tunnel-id

https://jenkins.opendaylight.org/controller/job/controller-merge/
    lastSuccessfulBuild/artifact/opendaylight/md-sal/model/model-flow-base/
    target/site/models/opendaylight-match-types.html
###########################
```

Appendix B

Example Topology Script for Mininet

This appendix includes the script used to build the Mininet topology used in the test bed.

mn−script.py

```
#!/usr/bin/python
#Library import
from subprocess import call
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSKernelSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.link import Link, TCLink
from mininet.util import irange



def federatedNet():
    net = Mininet( controller=RemoteController, link=TCLink, switch=
        OVSKernelSwitch )

    print "*** Creating Nodes ***"
    print "*** Adding remote controller ***"
    #Add a remote controller for MN to connect to
    c0 = net.addController( 'c0', ip='192.168.231.250', port=6633)

    print "*** Adding switches ***"
    #Add 5 switches
    switches = [ net.addSwitch( 's%s' % s ) for s in irange( 1, 5 ) ]
    print switches
    print "*** Adding hosts ***"
    #Add two hosts to the topology
    hosts = {}
    for h in irange( 1, 2):
        globals()['h'+str(h)] = net.addHost( 'h%s' % h, mac='
            00:00:00:00:00:0%s' % h, ip='10.0.0.%s/8' % h )
    #Configure switches for OF1.3 capabilities
    switches[0].cmd('ovs-vsctl set Bridge s1 protocols=OpenFlow13')
    switches[1].cmd('ovs-vsctl set Bridge s2 protocols=OpenFlow13')
    switches[2].cmd('ovs-vsctl set Bridge s3 protocols=OpenFlow13')
    switches[3].cmd('ovs-vsctl set Bridge s4 protocols=OpenFlow13')
    switches[4].cmd('ovs-vsctl set Bridge s5 protocols=OpenFlow13')
    print "*** Add links between switches ***"
    #Create links between switches
    net.addLink(switches[0],switches[1])
    net.addLink(switches[0],switches[2])
    net.addLink(switches[2],switches[3])
    net.addLink(switches[3],switches[4])
    net.addLink(switches[1],switches[4])
    print "*** Add links to hosts ***"
    #Add links to connect hosts to switches
    net.addLink(switches[0],h1)
    net.addLink(switches[4],h2)
    #Build and start the network
    net.build()
    net.start()

    CLI( net )
    net.stop()



if __name__ == '__main__':
    setLogLevel( 'info' )
    federatedNet()
```

Appendix C
XML Data from Validation and Testing

These listings are XML data collected during the validation phase. Parts of the XML response has been omitted for readability with a %[...] comment:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<table
    xmlns="urn:opendaylight:flow:inventory">
    <flow>
        %[...]
        <flow-name>openflow:2toopenflow:5IPto10.0.0.2</flow-name>
        <id>103</id>
        <table_id>0</table_id>
        %[...]
        <instructions>
            %[...]
        </instructions>
        <match>
            %[...]
            <ipv4-destination>10.0.0.2</ipv4-destination>
        </match>
    </flow>
    <flow>
        %[...]
        <flow-name>openflow:2toopenflow:1IPto10.0.0.1</flow-name>
        <id>102</id>
        <table_id>0</table_id>
        %[...]
        <instructions>
          %[...]
        </instructions>
        <match>
            %[...]
            <ipv4-destination>10.0.0.1</ipv4-destination>
        </match>
    </flow>
</table>
```

Listing C.1: Flow Rules on Node openflow:2 Before Move

```
No data exists.
```

Listing C.2: Flow Rules on Node openflow:3 Before Move

```
No data exists.
```

Listing C.3: Flow Rules on Node openflow:2 After Move

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<table
    xmlns="urn:opendaylight:flow:inventory">
    <flow>
        %[...]
        <flow-name>openflow:3toopenflow:1IPto10.0.0.1</flow-name>
        <id>102</id>
        <table_id>0</table_id>
        %[...]
        <instructions>
            %[...]
        </instructions>
        <match>
            %[...]
            <ipv4-destination>10.0.0.1</ipv4-destination>
        </match>
    </flow>
    <flow>
        %[...]
        <flow-name>openflow:3toopenflow:4IPto10.0.0.2</flow-name>
        <id>103</id>
        <table_id>0</table_id>
        %[...]
        <instructions>
            %[...]
        </instructions>
        <match>
            %[...]
            <ipv4-destination>10.0.0.2</ipv4-destination>
        </match>
    </flow>
</table>
```

Listing C.4: Flow Rules on Node openflow:3 After Move