NTNU
Norwegian University of
Science and Technology

Traffic Policing in Dynamic Military Networks Using Software Defined Networking

Hans Fredrik Skappel

Master of Telematics - Communication Networks and Networked Services
Submission date: June 2016
Supervisor: Øivind Kure, ITEM

Norwegian University of Science and Technology
Department of Telematics

NTNU – Trondheim
Norwegian University of
Science and Technology

Title:	Traffic Policing in Dynamic Military Networks Using Software Defined Networking
Student:	Hans Fredrik Skappel

Problem description

Software Defined Networking (SDN) is an approach to computer networking where the control plane and the forwarding plane is separated, and the control plane can manage multiple forwarding plane entities, located on multiple network nodes. SDN has received lots of interest and momentum in the last few years, where it is identified to offer more flexibility and control for the network administrators. OpenFlow (OF) is a protocol which is a part of the SDN suite, used for communicating between the control plane and the forwarding plane. Military networks have a strong focus on robustness and network utilization. These networks are dynamic in nature, where link capacities vary; both in time and per-link characteristics. Traffic in such networks must, therefore, be controlled in order adapt to present traffic and network conditions and while sufficient resources are available, traffic may be admitted. However, when there has been a change in the network, admitted traffic must be redirected and alternatively pre-empted in the case of insufficient resources.

Objective: The objective of this thesis is to explore SDN approaches on how to police traffic in networks, where both traffic, topology, and resources change.

Methodology: The candidate needs to identify how policing can be enforced in military networks, and show how SDN can be used as a tool for controlling traffic in dynamic environments. The candidate should also implement a proof of concept testbed utilizing SDN and OF for traffic control and policing.

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

Abstract

This thesis looks at how Software Defined Networking (SDN) can be used to provide traffic engineering and to police traffic in an Operational Military Network (OMN). SDN is a concept where the control plane is separated from the forwarding plane, and the control plane is capable of controlling forwarding plane elements located on multiple network nodes using the OpenFlow protocol. Specifically, we have discussed the problems in OMNs, and possible SDN approaches to mitigate the challenges. Based on the findings, we have designed, developed and validated an SDN implementation capable of obtaining dynamic topology information and to enforce user-defined policies in order provide traffic engineering for flows, resources, and topology.

Sammendrag

Denne masteroppgaven ser på hvordan Software Defined Networking (SDN) kan brukes til å utføre trafikkstyring og til å avgrense trafikk i et Operasjonelt Militært Nettverk. SDN er et konsept hvor kontrollplanet er adskilt fra videresendingplanet, og kontrollplanet er i stand til å styre elementer av videresendingsplan utplassert på flere nettverksnoder, ved hjelp av OpenFlow-protokollen. Spesifikt har vi diskutert problemene i OMNs og mulige tilnærminger med SDN for å mitigere utfordringene. Basert på funnene har vi designet, utviklet og validert en SDN implementasjon med evnene til å innhente dynamisk topologi-informasjon og til å håndheve brukerdefinerte regelsett for utøve trafikkstyring for trafikkstrømmer, ressurser og topologi.

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 (NTNU).

I would especially like to thank my supervisor Dr. L Landmark for helpful and thorough supervision throughout the project period. I would also like to thank Dr. M. Hauge and Professor Ø. Kure for ideas and inputs to my work.

Hans Fredrik Skappel
Trondheim, Norway
June, 2016

Contents

Appendices

List of Figures

List of Tables

List of Algorithms

List of Acronyms

API	Application Programming Interface.
ARP	Address Resolution Protocol.
CLI	Command Line Interface.
DPID	Data Path Identifier.
DTN	Deployable Tactical Network.
GUI	Graphical User Interface.
IDE	Integrated Development Environment.
IP	Internet Protocol.
ISP	Internet Service Provider.
LAN	Local Area Network.
LLDP	Link Layer Discovery Protocol.
MAC	Media Access Control.
MTN	Mobile Tactical Network.
OF	OpenFlow.
OMN	Operational Military Network.
OVS	Open vSwitch.
OVSDB	Open vSwitch Database Management Protocol.
PBR	Policy-Based-Routing.

PDP	Policy Decision Point.
QoS	Quality of Service.
SDL	Specification and Description Language.
SDN	Software Defined Networking.
SMCN	Strategic Military Core Network.
SNMP	Simple Network Management Protocol.
STP	Spanning Tree Protocol.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.
UDP	User Datagram Protocol.
VM	Virtual Machine.
WAN	Wide Area Network.

Chapter 1 Introduction

An Operational Military Network (OMN) is a network used for military purposes and needs, and can be characterized by having strict requirements for availability and performance. OMNs are used in operations where the network conditions may experience variations, both in forms of resources, topology, and traffic. The key element in maintaining a sustainable quality of the network is to perform traffic engineering and policing in a way that utilizes the network resources in the best possible manner, and so that high priority traffic has sufficient capacity. This, however, can be challenging to manage in complex networks where the need for functionality forces different technologies to cooperate.

Software Defined Networking (SDN) is the term where the control plane is decoupled from the forwarding plane and placed in a controller [22]. An SDN controller can manage the forwarding tables located on multiple SDN switches by using the OpenFlow (OF) protocol. In later years, there has been an increasing movement towards software-defined approaches in general, where SDN is the software-defined approach to networking. SDN is believed to offer flexibility and new opportunities of programming the network for the administrators. Instead of being dependent upon using several vendor-specific technologies, the administrators can develop their platform with features and functions uniquely customized for the network.

1.1 Motivation

OMNs are networks used for communication in military operations on a worldwide scale. The networks design and structure may vary from nation to nation, but they essentially share the common factor that significant parts of the transferred information may be of high importance for the operations. Some of these networks should be deployable in operational areas, which includes countries lacking proper network and power infrastructures, such as Afghanistan, of which implies that the networks can not be dependent on having unlimited capacity. The probability of resource limitations in the networks, are therefore present, and may in worst case affect the operation. Communication between dynamic units out in the field is mainly radio [41]. In other words, the risk of delay, loss and disconnection are high, which overall implies that the networks have many vulnerable factors. Despite this, it is essential that these networks are fully sustainable during ongoing operations, even though the network resources may vary or be limited. High requirements for network utilization and traffic control, to ensure utilizing the resources in the network in the best possible way without causing restrictions to ongoing operations.

Using the SDN technology in an OMN may advantageously achieve flexible and new ways of ensuring network utilization, based on documentation about SDN and previous work in [30]. The motivation for this thesis would be to look at SDN as a tool for traffic policing and traffic engineering in OMNs. It is of interest to explore approaches and capabilities by using SDN, and to discuss the findings.

1.2 Derived Problem Description

Based on the problem description earlier presented, the derived problem description is stated as following:

How can SDN be used for traffic policing in an Operational Military Network? A laboratory SDN application should be designed, developed and validated to show how traffic engineering and policing can be conducted using capabilities from the SDN suite.

1.3 Outline

The thesis is structured as follows:

- **Chapter 1** is an introduction to, and motivation for, this thesis.
- **Chapter 2** presents relevant background information on the subject.
- **Chapter 3** is a discussion of SDN in Military Networks and approaches to traffic engineering, based on presented information from Chapter 2.
- **Chapter 4** will present design requirements and decisions for the laboratory implementation.
- **Chapter 5** presents background and technical information about laboratory implementation.
- **Chapter 6** is a validation of the SDN application in various scenarios.
- **Chapter 7** will present practical solutions and experiences gathered through the implementation work.
- **Chapter 8** is a discussion of the total work, based on background information, gained experience and validation results.
- **Chapter 9** contains the thesis conclusions and further work.

Chapter 2 Background

This chapter will provide the theory which is a part of the scope of the laboratory implementation. The implementation includes an SDN controller with the use of OF 1.3. Background information regarding Operational Military Networks is presented to give the reader an overview of important characteristics and aspects which are essential in these networks. An introduction to features included in the traffic engineering context is also presented. Parts of the background are obtained from previous work in [30].

2.1 Software Defined Networking

SDN is a network architecture where the control plane[1] and the forwarding plane[2] are physically separated and where the control plane controls several devices [11]. In regular legacy routing, both planes are typically located on the same device, which makes the device (i.e., a router) self-depended and capable of making its decisions. Following this, having the control plane embedded within the device can introduce challenges in forms of management. An example is to handle software updates or configurations related to the control plane of the device. Due to the embedded control plane and/or by using equipment from several vendors, it may be times where the management needs to be done separately on each device, either in forms of remote connection or by physical presence. This is a job that could be time demanding in a large networks. SDN mitigates these challenges by having a single or a few control plane entities which interact and manages several switches remotely, thus simplifying network configuration, updates, and customization. Instead of updating all control plane devices, it may only be necessary to update the controller.

In SDN, the control plane is placed in a controller, usually centralized, which has secure encrypted connections (mainly Transport Layer Security (TLS) [15]) with the

[1] The collection of functions responsible for controlling one or more network devices [18].

[2] The collection of resources across all network devices responsible for forwarding traffic [18].

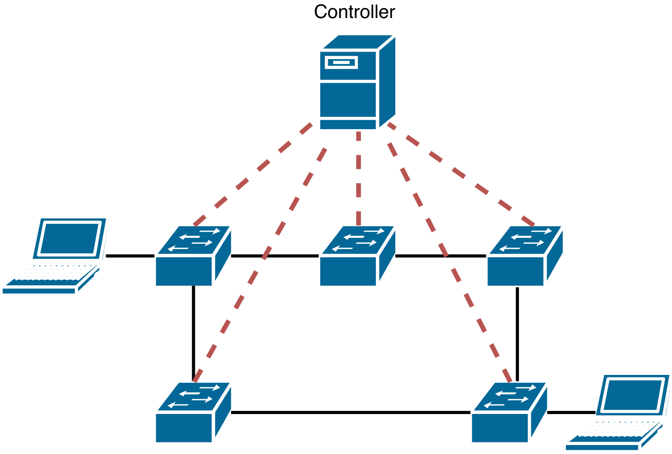


Figure 2.1: SDN controller

corresponding switches. SDN is a concept, and does not specify how to communicate with the network devices, but the most used protocol is OF. The SDN concept is illustrated in Figure 2.1, where the red dotted lines illustrate the secure links which the controller uses to communicate with the switches, described in details in Section 2.2. The lines connecting the switches are links used for network traffic.

SDN is an overlay architecture, meaning that it is dependable of having an underlying technology to work. To boot up an SDN network, the technology is reliable of having a logical link with Transmission Control Protocol (TCP) / Internet Protocol (IP) connectivity between the switches and the controller. SDN is independent of what kind of technology the logical link may exist of; whether it is a physical link or a path with several routers does not matter. Firstly when the controller and the switches establish the connection, then a transition to using SDN can be executed. The traffic will, nonetheless, still be transported by the underlying nodes throughout the period. The controller will only see the SDN switches from its viewpoint as Figure 2.2 illustrates, where the flows are physically forwarded through a path with three legacy routers, but the controller only sees the logical link. This viewpoint also applies to the management links connecting the controller with the SDN switches. [17]

SDN introduces the expression *flows*, which is a flexible way of forwarding streams of packets based on various properties from the header. Legacy routing relies entirely on the destination IP address, while in SDN, forwarding decisions can be performed based on many other properties such as protocol, type of service, port, or combinations of these. A forwarding condition can apply to many streams of packets; thus, a flow. An example of this is to forward User Datagram Protocol (UDP) traffic one way while TCP traffic another way, independent of source or destination addresses. This indicates that TCP traffic and UDP traffic are two different flows.

The SDN controller is typically the only entity with a state in a traditional SDN network. The SDN switches are dumb in the sense that they must be instructed by the controller what to do. All forwarding devices within an SDN environment are commonly referred to as switches or SDN- or OF-switches; there are no routers. The actual routing takes place within the controller application, where the application will find a suitable path in the network, and then install flow table entries[3] on the switches along the path so the flows can be forwarded correctly. All switches possess a flow table, which is where the controller can install flow table entries which instruct the switch where to forward the flows. Forwarding is performed by the switches, based on inspecting their flow tables when incoming packets arrive. If the switch finds a match in the flow table, the packet will be forwarded according to the flow rule action. If an incoming packet does not match with any flow rules in the flow table, then it will traditionally be forwarded to the controller. In this way, the controller is always notified when new incoming packets arrive in the network.

Forwarding rules can be inserted either in a proactive or a reactive manner. When using a proactive approach, the rules are installed on the switches before packets arrive. The reactive approach install rules on the switches as the packets arrive the network: when the packet enters the network for the first time, it will not have a match with any flow rules, and therefore will be forwarded to the controller for processing. There is also a possibility to combine these approaches to make a hybrid solution.

The SDN controllers typically have two interfaces which are called the northbound and the southbound interface [22]. The northbound interface is an interface between the application plane[4] and the control plane, while the southbound interface is defined as the interface between the control plane and the forwarding plane, which is where the OF protocol is used. Applications at the application plane communicate through the northbound interface and is software that utilizes underlying services to perform a function [18], thus being the plane where end users will program the controller to manage the network entities[5]. The applications use functionality from the southbound interface to communicate with the switches, such as example various OF methods to fetch switch statistics. Some controllers may also have a third interface called the east-west interface, which multiple control plane entities uses for communication between them.

[3] *Flow table entries* are also addressed as *flow rules* in this thesis.

[4] The collection of applications and services that program network behavior [18].

[5] In the rest of this thesis (unless specified otherwise), referring to the controller also includes the application that manages the controller.

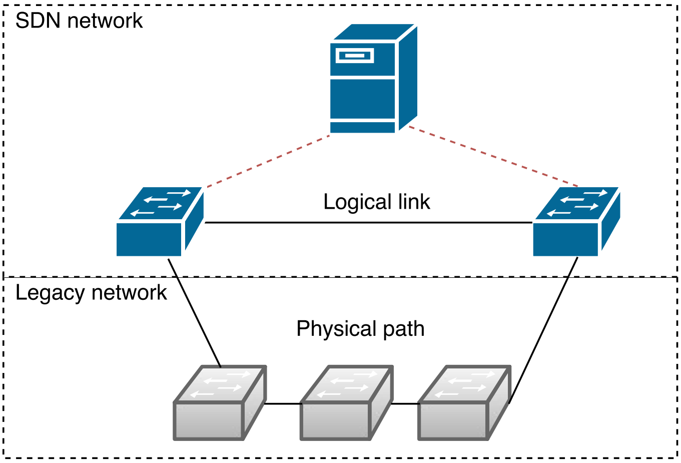


Figure 2.2: SDN as overlay network

As today, there exists a variety of SDN controller frameworks for developing SDN applications. The revolution with such frameworks is the open access to the control plane, by which the network resources can be programmed and customized to perfection without being limited by using vendor specific software or protocols. Instead, the applications can use the controller's southbound interface to manage several switches, of which contributes to ease the management by ensuring configuration consistency.

2.2 OpenFlow

There have been several types of research on programmable networks throughout history, whereby the subject has received more attention and momentum since the release of OF [11]. OF is the first standardized communication interface defined between the control and the forwarding layers of an SDN architecture [16], previously pointed out as the southbound interface. The OF protocol includes the necessary features and messages to use a controller to manage the forwarding plane within an SDN environment. To use OF, it must support both sides of the southbound interface, meaning both by the SDN controller framework and the switches. Figure 2.3 illustrates the OF communication between a controller and a switch. The controller and switch communicate through the secure channel, while the controller can modify the flow and group tables entries within the switch.

The secure channel from the controller to each switch is also known as the OpenFlow channel and is used by the OF protocol to exchange management traffic. The

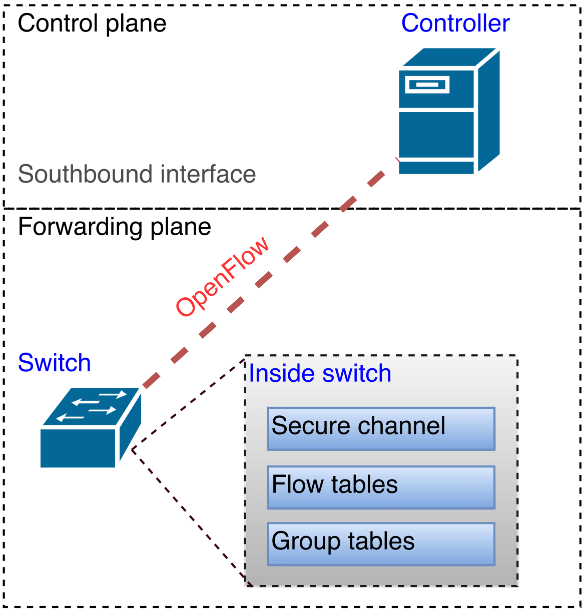


Figure 2.3: OpenFlow

channel can be both in-band[6] and out-band[7], meaning it necessary does not need a physically separated connection with the controller, the only requirement is that it should provide TCP/ IP connectivity [15]. Due to SDN and OF is an overlay technology, it would imply that there needs to be legacy IP routing enabled at the bottom to forward the management traffic between the switch and the controller [21]. At boot-up of an OF network (enabling the SDN overlay), the switches must be user-configured to know the IP address of the controller before the establishment of the connection. The management traffic is not processed by the OF pipeline[8], and is usually communicated through a physical out-band management port. If TLS encryption is applied, the communicating parties need to exchange certificates, and this must be user-configured as well. When the OF establish the connection, the switch and the controller exchanges *OFPT HELLO* messages [14], where the version field set to the highest OF version supported by the sender. If successful, the OF connection proceeds and the controller can start instructing the switch using the highest protocol supported by the mutual entities. [15]

For the administrator, OF is mainly used to access the forwarding plane of the connected switches to install forwarding rules. Each switch has flow tables and group

[6] Management traffic using the same connection as regular traffic.
[7] Management traffic using a separate connection.
[8] The OF pipeline is the matching of the packet against flow and group tables.

tables which can be populated and modified by the controller. OF uses the concept of flows to identify network traffic based on predefined match rules that can be statically or dynamically programmed by the SDN control software [16]. A flow rule consists of match conditions and actions, which provides a forwarding policy for a particular flow. The flow table also keeps track of statistics by counting the number of packets and bytes associated with a flow, as well as a counter which keeps track of the time since the flow rules were used [15].

Groups are abstractions which are used to represent a set of ports as a single entity for forwarding packets, such as splitting traffic. Each group is composed of a set action buckets, whereby every bucket contains a set of actions to be applied before forwarding to the port(s). Action buckets can also forward to other groups, enabling to chain groups together. The chaining of tables can also be applied to regular flow tables, to offer flexibility to the packet processing pipeline. As the packet traverse through the pipeline, a packet is matched and processed in the first table, and may be matched and processed in other tables. Table 2.1 and 2.2 displays the various fields of which a group entry and a flow entry exists of.

Table 2.1: Group table entry [15]

Group ID	Group Type	Counters	Action Buckets

When a packet arrives at an OF switch, the switch will look for a flow rule which matches parameters from the packet. By matching with the flow rule, the switch will forward the packet according to the instruction defined in the flow rule. If the packet does not match with any flow table entry, then the packet will either be sent to the controller or be discarded, depending on the network configuration. In the legacy version of OF, the default action is to forward the packet to the controller over the OF channel [23]. The controller will then be able to process the packet, which ensures that the controller will always get the packets in scenarios where the switches do not know where to forward them. When incoming packets arrive into the network, and they match with a flow rule, they will be forwarded by the switches without notifying the controller. Figure 2.4 illustrates the forwarding based on flow table lookups: if the flow matches with a flow rule condition, then the flow rule executes the corresponding action. The packet will be forwarded to the controller if the packet does not match with any rule.

Table 2.2: Flow table entry [15]

Match Conditions	Priority	Counters	Timeouts	Cookie	Instructions

Packets sent to the controller can be processed on the control plane. The controller will typically inspect the packet and takes a forwarding decision based on its knowledge

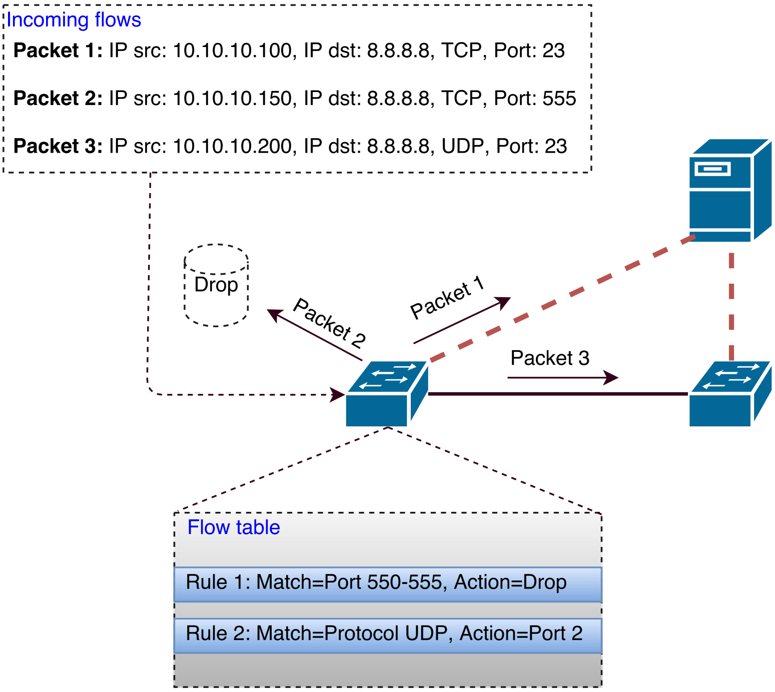

Incoming flows
Packet 1: IP src: 10.10.10.100, IP dst: 8.8.8.8, TCP, Port: 23
Packet 2: IP src: 10.10.10.150, IP dst: 8.8.8.8, TCP, Port: 555
Packet 3: IP src: 10.10.10.200, IP dst: 8.8.8.8, UDP, Port: 23
Drop
Packet 2
Packet 1
Packet 3
Flow table
Rule 1: Match=Port 550-555, Action=Drop
Rule 2: Match=Protocol UDP, Action=Port 2

Figure 2.4: Flow table lookup

of the network topology. The controller will then install flow table entries to the switches along the path, and the switches will then forward the packets. An example of an OF flow rule installation is illustrated in code sequence 2.1. The code sequence defines *Rule 2* in Figure 2.4. The sequence instructs the controller to send an OF Flow Modification Message[9] [13] to a switch with a match condition and an action: the switch will forward packets with protocol number 17 (UDP) out port 2. The last line in the code snippet sends the message to the switch, ensuring that the switch installs the flow rule.

Code sequence 2.1 Flow rule installation using POX controller.

```
msg = of.ofp_flow_mod()
msg.match.nw_proto = 17
msg.actions.append(of.ofp_action_output(port=2))
event.connection.send(msg)
```

[9] OF Flow Modification is an instruction used to modify or to install flow rules on a switch.

2.3 Military Networks

Military networks share the same functions as civilian networks, but the difference lays in the network's purpose, by which the military network serves a military purpose, where it follows various other considerations. A large-scale civilian network (e.g., Internet Service Provider (ISP) network) typically has a static infrastructure with a lot of capacity, while the military networks are characterized by having a more dynamic nature due to the use of heterogeneous bearers because of their functions (i.e., deployment requirements). Where the civilian networks use fiber, the military networks may need to use radio link or satellite, which introduces capability limitations as well as a higher probability of errors and variances.

Operational Military Networks varies from small national Local Area Networks (LANs) to bigger Wide Area Networks (WANs) shared with several nations. Therefore, it is hard to give a clear definition of what an OMN is. OMNs can carry classified information that is critical, and it is essential that the availability remains high at all times [34]. In military operations, the networks can be used to exchange orders and operational decisions, in which requires immediate responses and actions. In these situations, the network needs to be functional, and the network resources should have the necessary capabilities to transfer the information across the network.

The conventional OMN characteristics are the requirements for flexibility and robustness [38]. A significant difference between a military and a civilian network could be the potential threat from attacks, whereby the OMNs, in general, should be able to resist or survive attacks, both in forms of physical attacks on infrastructure and cyber-related attacks. Physical assault, e.g., bombing a fiber link, could tear down the network and paralyze an entire operation. Notwithstanding, the network should work in dynamic and fast changing environments, by reacting and adapting to sudden infrastructure or capability changes. To cope with these changes, a strategy would be to implement traffic engineering mechanisms of which adjust traffic and resources accordingly.

A significant proportion of the traffic in OMNs may typically exist of traffic of higher importance than other. Consequently, are service differentiation and Quality of Service (QoS) important attributes in an OMN, to prohibit the high priority traffic to suffer. In a network with limited capabilities and high exposure to variances, there must be mechanisms implemented to ensure that the prioritized traffic gets free resources [41]. This is where an OMN differ from civilian networks, because there is usually not an opportunity to buy more capacity if the network is already deployed in a military operation. For example, it is not possible to achieve the same bandwidth capabilities as a fiber link if the operation is limited to using satellite communication because of its remote deployment. The emphasis is to utilize the

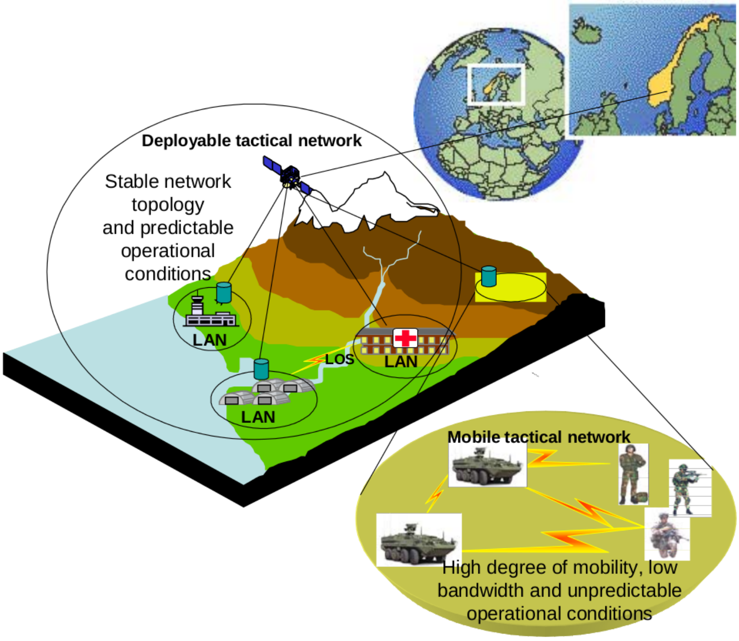

Deployable tactical network
Stable network topology and predictable operational conditions
LAN
LOS
LAN
LAN
Mobile tactical network
High degree of mobility, low bandwidth and unpredictable operational conditions

Figure 2.5: A three-level military network topology [41]

accessible network resources in a smart way.

As previously presented, there is no standardized definition of what a military network is, and the military networks will most likely differ from nation to nation. This thesis defines three main types of OMNs. The networks are connected but serve different functions in an operation. Figure 2.5 illustrates the connection of the three networks.

There are Strategic Military Core Network (SMCN) which have similarities to traditional private carrier backbone networks. The SMCN is deployed in the country of the military nation and has the primary function to be the national backbone to connect national forces and other civilian departments [31]. A SMCN typically cover large geographical areas, due to that the national military is nationwide. The core infrastructure typically consists of several separated networks which carry different national classifications, due to security regulations. This network is usually more static than the other OMNs, due to it being the core network using a fixed infrastructure. The network is assembled by using various transmission bearers, such as fiber, radio links, and satellite to interconnect to other remotely deployed networks [41]. The networks may also be overlay networks built on top of a civilian core network.

The Deployable Tactical Network (DTN) is the network which is deployed as a part of international or national operations. It will typically be an interconnected network where allies communicate with each other in the mission area [31]. The purpose of the network will be to provide local connectivity at the operational site by serving as the local temporary backbone network with connections to Mobile Tactical Network (MTN) and SMCN. A coalition network often is composed of a heterogeneous structure, due to various transmission bearers and equipment. However, the network may be considered as stable because the placement of its core elements is essentially inside the military camps. Due to the networks remote deployment, the communication with the SMCN may be via satellite link or tunneled using the country's infrastructure.

The Mobile Tactical Network can be defined as the network used in mission operations outside of the camp. An MTN is characterized by existing of mobile nodes, in an environment which is dynamic and where the communication is mostly based on radios [41]. The network resources are limited regarding power, resources, range and delay because the network entities can vary from small hand-held devices to vehicle-mounted solutions. Seeing that the military units are constantly moving, the ability for ad-hoc routing [8] and relaying traffic may be present to cover the communication area. Significant parts of the data are real-time based, (i.e. audio and video) due to the purpose of the MTN is to serve as a transmission bearer between units in ongoing operations.

2.4 Traffic Engineering

Traffic engineering is defined as that aspect of network engineering dealing with the issue of performance evaluation and performance optimization of operational IP networks [2]. It addresses the challenges concerning efficiently allocation resources which are beneficial for the users, and can be performed automatically or through manual intervention. Traffic engineering use methods such as admitting, abstraction, blocking, re-routing, queuing, preempting, policing or shaping, which are different ways to adjust and customize the network traffic according to desired behavior. One of the most important features performed in networking is routing of the traffic. As a result; one of the most distinctive functions conducted by traffic engineering is the control and optimization of the routing function, to steer traffic through the network in the most efficient manner [2].

2.5 Policing

Policing is a feature which ensures that traffic does not exceed certain threshold limits. Policing of a stream is when the traffic controlling entity starts dropping or discarding packets from the stream to bring stream into compliance with a traffic profile [3]. The threshold limits can either be predefined by the administrators or dynamically defined based on feedback from the network.

Policing can be used to prevent the networks from suffering. If the transmitting traffic (i.e., UDP) exceeds the network's maximum capacity, it will lead to network congestion and possibly send particular network devices into fault state, but policing can prevent this from happening. There is also a possibility to police only parts of the traffic or specific flows, which can be necessary for networks where certain traffic has priority. In SDN setting, the controller may appear as the natural entity to take the role as the traffic controlling entity. However, the policing rules can be installed at the switches by the controller, and be enforced by the switches when the traffic arrives [39].

2.6 Policies

Policies can be considered as a part of the traffic engineering context. A network policy can be defined as a set of policy rules which are managed by the administrator, used to control the resources[10] in a network [32]. The network will provide services about the defined policies, which essentially means that policies define the network behavior. The network behavior is the relationship between the clients or services using the network resources and the network elements that provide the resources.

A policy will typically consist of one or several conditions and corresponding actions as Code sequence 2.2 presents. Actions are linked to either meeting or not meeting a set of match conditions defined in the policy. In other words, a policy specifies what action(s) must be taken when meeting a set of associated conditions [32].

There are two ways to trigger a policy, which is either statically or dynamically. Static policies apply to a fixed set of actions in a predetermined way according to a set of predefined parameters that determine how to use the policy [33]. Therefore, static policies state how to use the resources, independent of the dynamic feedback from the network. An example of a static policy is to deny access to network resources for a particular IP address.

Dynamic policies are triggered when needed, based on a changing condition. For instance packet loss or congestion. The key element in a dynamic policy is to obtain network parameters, of which the policy can use as an input. Verifying of the dynamic policies are continuous, so whenever there is a change, it can be verified against the policy. About obtaining dynamic data, it is necessary monitoring mechanisms which passively listens or actively requests for network information.

[10] A physical or virtual component available within a system [18].

The policy applies to different levels. Examples are user level, where a particular node or user is affected by the policy, or service level where an individual service is impacted. A policy can be very specific, meaning that it can be applied to a single level, but can also be broad and include elements from many levels. In this manner, match conditions can be customized and narrowed down. There are also variations on how the policies are applied; a policy can, for instance, be strict or preferring. An example of a strict policy is to deny access to new connections in a congested path, while a preferring policy is to try to find the least congested path.

Code sequence 2.2 Policy structure when using conditional statements.

```
IF <condition n>... AND <condition n+1 >
THEN <action m>... AND <action m+1>
```

Routing based on network policies can be referred to as Policy-Based-Routing (PBR) [5] in legacy routing. Routing based on policy implies that the packet is checked against a list where the policies are placed. If the packet corresponds to a particular flow, then it will be routed according to the action defined in the policy, thus overriding the normal routing procedures [17]. SDN have similarities with PBR on forwarding level, due to the use of flow rules which instructs the switches where to forward the traffic by using various parameters from the packet. It is, however, up to the developers of the SDN application to make the policy environment to check incoming traffic against a policy list.

Policies in general, are ways of expressing the rules of the network, which can be used to define desired network behavior by administrators. Without being able to set rules and manage the network, the general network performance will probably suffer when exposed to variances. Using policies enables the administrators to specify how to use the policies, how the policies are triggered, and at which level to apply the policy [32].

2.7 Quality of Service

Quality of Service is the collective term for mechanisms that are used by the service provider to prioritize traffic, control bandwidth and control network latency [40]. QoS has similarities with traffic engineering, but focus on the achieved service quality, the ability to provide different quality and to guarantee certain levels of performance to the flows.

QoS can be used to provide traffic priority to particular flows based on various parameters from the packet. For example to ensure that an individual service always has the required bandwidth, and or does not exceed the maximum error rate. In an

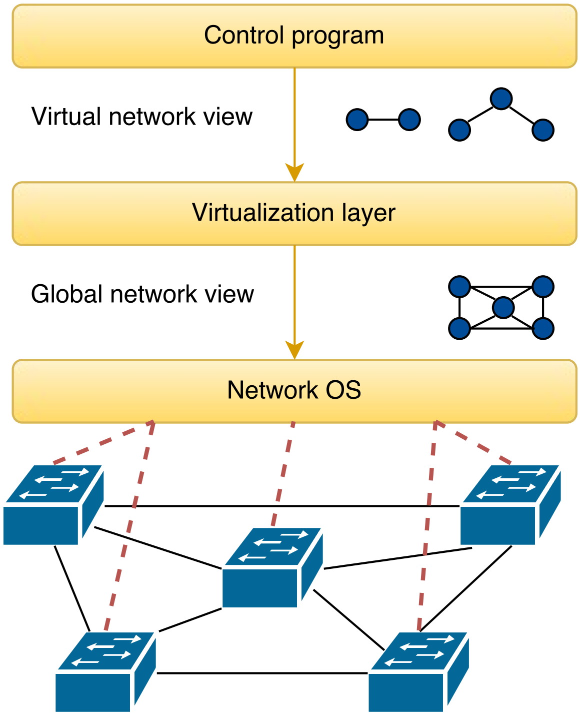


Figure 2.6: Abstraction layers [6]

SDN environment, QoS can be achieved by implementing various traffic engineering mechanisms, such as resource reservation, policing and queuing.

2.8 Topology Abstraction

Topology Abstraction is a way of manipulating top layers abstract view of the network [29] and is used to separate concerns in network environments. It is not necessarily a good thing to work directly on the physical network topology, due to the low-level complexity. By introducing different topology layers, they can concentrate on individual tasks, rather than a big process performing every task. For example, software programmers rather use Java [20] to develop their projects rather than of low-level languages. Instead, they compile the Java code into low-level instructions, and the same thing applies to networking.

The Internet architecture is layered according to the TCP/IP model [1], where the different layers forward the traffic based on various factors which vary from each layer. Examples are the link-layer which does forwarding from link to link based on Media Access Control (MAC) addresses, while the network-layer uses IP addresses to route packets on paths composed of several links. By introducing abstraction layers, they can all serve a different purpose, while interacting with each other. The layers will have a different understanding of how the network components are assembled.

Figure 2.6 illustrates a topology abstraction with several layers using an SDN approach. The top layer express the desired network behavior based on an offered a virtual network view from the lower layer. The upper layer does not need to know what kinds of actions are taken on the lower level, which can be regarded as a way of hiding unnecessary information. The virtual layer is built on the global network view, while the network operating system (SDN controller) controls the switches. Therefore, the different layers serve a different function, while combined makes the system complete. The advantages of this approach are to separate responsibility, hide complexity and to speed up processing by letting the layers work simultaneously.

The essence is that topology abstractions can apply to different things in the network, such as to abstract routes, links, switches, functionality, and management. Especially considering SDN, topology abstractions becomes important due to the open access to the control plane. The developers would need to create functionality working on global level all the way down to low-level; from generating paths, to controlling where to steer the flows on a port-level basis. It would be beneficial to automate this process and place functionality in separate modules and to let other modules (using different topology abstractions) use their functionality. Such as various topology modules that expresses virtual paths based on abstracted views of the topology, while a low-level module uses the inputs to installs the flow rules on port or link basis. Extensive information regarding approaches to topology abstractions in SDN are found in [29].

Chapter 3
SDN in Military Networks

This chapter is a discussion where earlier presented background information from Software Defined Networking and OpenFlow is merged with the Operational Military Networks requirements and characteristics. The goal of the discussion is to identify if SDN can fulfill the military demands by performing traffic engineering.

3.1 Motivation

Operational Military Networks are complex with high requirements to performance, availability and survivability. Primary legacy routing is not always the efficient way, and these networks are dependent on traffic engineering to fulfill requirements. In addition to the requirements are the vulnerable factors which also needs to be taken into consideration, because the networks can be exposed to a variety of events and incidents which may not occur in civilian networks. The combination of strict requirements to performance and availability, combined with the exposed weaknesses makes these networks unique. That said, SDN is believed to offer sound ways of doing traffic engineering; mainly due to the administrators opportunity to program the network resources to fit the individual network.

However, will traffic engineering with SDN be the answer to the challenges the Operational Military Networks are facing, and to what extent would it be realizable to deploy?

3.2 Challenges in Military Networks

As previously pointed out, it is the combination of high requirements and the exposure that makes the different OMNs complicated. There are various network factors which may affect the OMNs availability and performance, and an assortment of factors is listed below.

- **Capacity limitations**. Due to using heterogeneous wireless bearers, the bandwidth will most likely vary and have limitations.
- **Delay**. Limited bearers, such as satellite, introduce long delays. Traffic may also be relayed or repeated if military units are far away from each other.
- **Jitter and noise**. Jitter and noise will impact the networks used in dynamic operations. Terrain and weather will affect the signal.
- **Signal strength** will vary as a result of moving units and may in cases lead to temporary disconnections or an increased error rate.
- **Component failure**. The network equipment can fail due to the exposure to extreme weather conditions because of the geographical operation area. Battery powered equipment may be drained and go offline. The risk of potential enemy attacks is present in both physical and cyber domain.

The various OMNs are more or less exposed to variances where the listed factors are influenced in some way, but the three networks will experience differences. The Strategic Military Core Network is less affected by variations and limited resources, and it is most likely the satellite communication interfaces towards the other networks which are the weakest links regarding limitations and weaknesses. The network is more of a static network with minor changes, although heterogeneous bearers will ensure variances regarding bandwidth and performance. Failures occurring in this network are exceptions.

In Mobile Tactical Networks, failures and variances are not exceptions, but they are instead more or less expected because of the network's structure. This network is heavily influenced by a changing topology, due to the use of mobile nodes and wireless bearers. The devices are in practice located with dynamically moving military units, such as radios used for voice communication between the groups. Nodes can be widely spread, and it may be necessary to relay traffic between them to communicate with a particular station, thus the expose of delay and variances.

The Deployable Tactical Network is more vulnerable to limitations than the SMCN, but not as much as the MTN. The DTN plays the role as the intermediate strategic network backbone within the operational area. Due to the network's remote and temporary deployment, a significant use of wireless bearers is implied, with interfaces to both SMCN and MTN, which leads to that the network also has to deal with the weaknesses that accompany these interfaces. The network's role stresses the requirement for availability, due to it is serving as the gateway into the base for MTNs, whereby a possible purpose would be to provide additional military support in critical operational situations [41].

The presented information indicates that all the Operational Military Networks this thesis define, are facing challenges when it comes to their requirements and the exposed weaknesses. Although, their structure, purpose and usage separates them and ensures that they are facing different challenges.

3.3 Coping with Challenges

Traffic engineering becomes necessary to be able to cope with the challenges of which the various OMNs are facing, by utilizing and controlling the network resources in the best way possible. As earlier stressed, the network resources within a military network, especially regarding MTNs can be limited, due to using it in field operations. The topology includes the usage of mobile nodes, and is therefore dynamic in the sense of nodes come and go. If a sandstorm occurs in the operation area, it could tear down the wireless network communication by affecting the signal strength as well as the jitter and noise, leading to an immense error rate. If a military vehicle drives on the other side of a mountain, it may result in blocking of the radio signal, leading to a temporary disconnection, thus the dynamic factor. The point being that the network exposed to several weaknesses, which stresses the need for robust mechanisms which can withstand and make adjustments when the events occur.

To mitigate the limited resources in remotely deployed networks, there is usually not a realistic approach to only increase capacity. Instead, the emphasis should also be to utilize resources. A solution to the problem would be to implement policies, which are used to link conditions with instructions to define behavior in various situations. This implies that it is of high importance that policies are of a dynamic character because the policy should trigger on changing conditions. The primary objective would be to generate the best outcome based on the attributes of which are considered essential for a particular flow, service or network itself. In some scenarios, the delay may be the most important quality, sometimes bandwidth, depending on the circumstances.

The key element in conducting traffic engineering in such network environments will, thus, be to strengthen the ability to cope with the changes and dynamically occurring incidents. This would require continuously monitoring of the topology as well as the flowing traffic within the network. A non-working operational network can in worst case have fatal consequences when used in operations, which, again, emphasizes the difference between a civilian network and a military network.

3.4 Designing SDN for Military Networks

An implementation of SDN would require changes to the military network design. OF switches would replace legacy routers and switches, and also to move away from a decentralized control plane to a centralized controller. Nevertheless, it is a possibility use SDN in only parts of the networks; it is not necessary to change everything. Most importantly, end-clients will not notify the difference between a legacy network and an SDN network, since the modification will only influence the forwarding nodes in the network.

The placement of the SDN controller is essential to achieve the desired traffic control. The controller should not be controlling too many entities because this will generate a lot of traffic and processing at the controller. Bandwidth can be a limitation in the OMNs, and it is of interests to send as minimum management traffic as possible without causing restrictions. In large-scale networks, a solution could be to use multiple controllers which only control a portion of the entire network or subnets. Regarding possible SDN designs, there are several studies on the topic, and a thorough review for military networks is documented in [35]. The essence being that when the networks grow, it may be necessary to use several controllers to divide areas of responsibility while exchanging information with each other, thus require a scalable design.

By implementing SDN, the network must achieve just as good or better security than the today's networks. The SDN controller is the single-point of entry when new packets arrive as well as the central controlling network entity, which makes it one of the most important nodes in the network. The controller is an attractive target for a potential enemy, but also vulnerable to occurring link failures. In a traditional simple SDN network, the controller will in addition be the single-point of failure, but this can be mitigated by implementing fail-over solutions by installing backup controllers, or legacy routing procedures take over if the controller is disconnected [37].

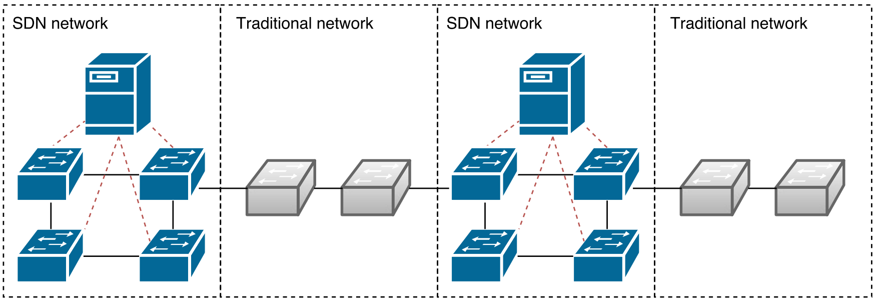


Figure 3.1: SDN and traditional networks combined

Figure 3.1 illustrates a network existing of SDN and legacy subnetworks. This implies that SDN can be deployed only to the networks where it is beneficial to implement SDN, or to subparts of the networks. Legacy routing combined with traffic engineering can be good enough in most situations, and many times even a better solution because of autonomous routers and the decentralized control plane. The key element is that SDN must be designed differently based the challenges and requirements of which the network is exposed to. SDN does not solve every network problem and is also facing many of the same problems that exist in traditional legacy networks. This thesis addresses the following as the most important design factors regarding SDN:

- **Management Connection:** must be in place for the controller to instruct the switches. Limitations and variances of this connection must be considered. A centralized control plane only appears as a robust approach when management links are stable. The design must include backup strategies such as redundant controllers or functionality to perform changeovers to legacy routing. The placement of the controller and the communication between controllers are also parts of this context.
- **Controller Processing:** an extensive network with a significant amount of traffic, nodes, and variances will affect the processing needed by the controller. The network must be designed, so the load on the controller is acceptable, but also scalable if the network should grow over time. To include comprehensive functionality and detailed control (i.e. traffic engineering features) into the application will probably require the controller to do more processing and keep more state about the network.

The essence is to identify where SDN can be used and where it has its significant benefits regarding the presented OMNs. Based on earlier presented background information, it is possible to discuss the networks requirements and characteristics with SDN:

SDN in SMCNs emerges as a sound way of doing forwarding of traffic, because of the network is static in nature and transports various traffic, of which advantageously could be sorted into flows. The amount of traffic and the network size emerges as the determine factors when designing the network because there will most likely be necessary to include multiple controllers to the design. The static structure ensures that management links are robust, so a centralized SDN design emerges as a good option.

SDN in MTNs appears as a good match by looking at requirements and traffic related challenges. Variances in resources and topology are expected, and the network's structure includes limitations in forms of capacity, so it would be beneficial to have the flexibility of program the network resources for utilizing purposes. On the other hand, connectivity variances are addressed as a major challenge concerning management, because of all the issues that follow the disconnection of the SDN controller. When a disconnection of a management link occurs, it could be necessary to swap over to legacy routing approaches, but constantly disconnecting management links would then lead to regular changeovers, which is not efficient in the long run[1]. The network may end up with using legacy routing more than SDN. There might also be issues with long delays because of the possible use of relayed management traffic, which can introduce additional drawbacks. Possible designs could be to use one controller per switch, or a clustering approach where the controller only control a small share of the network, where the switches in this share are located physically close to the controller.

SDN in DTNs stands out as a potential approach based on the previously presented background information. DTNs introduces capacity limitations, and variations in topology and traffic will influence the network. The networks also have strict requirements for availability and performance, due to its important responsibilities and tasks in the operation. The DTN is a semi-static network, but with dynamic factors, whereby it is not as heavily influenced by changes and limitations as the MTN, but more than the SMCN. By focusing on the parts of the network where the need for traffic engineering is present, it would be a great advantage to be able to program the network resources to do specific packet handling. A traditional SDN approach with a centralized controller with responsibility for a particular share of the network could emerge as a possible strategy.

It appears as SDN can be implemented in all of the networks, but the design would likely vary to fit the different networks. The networks share the same essential requirements, while their characteristics differ more. That said, all the networks would require secure and robust designs regarding of management links controller processing. SMCN is just slightly affected by changes and limitations, so the use of SDN would mainly focus on the forwarding, but would offer flexible ways of separating traffic into flows. The use of SDN in core networks is well documented and already in use by large enterprises, such as Google's B4 [19]. MTNs represents the opposite case, where the variations can be extreme, and the resources are limited. The changes would most likely cause severe problems for the management of this network, so this would need high requirements to the design. Especially considering that the use of multiple SDN controllers would add significant complexity to the design and development process. DTN is the intermediate network, where limitations and variances are present, but not as extreme as in MTNs and not as low as in SMCNs. The network complexity and required functionality would likely correlate with the exposed variances. Regarding functionality, it is, therefore, likely to state

[1] If management traffic with one switch goes down, then the controller should also instruct all the other switches in the network to go over to using legacy routing in order to prohibit potential loops.

that the MTN would require the most, SMCN the least, while the DTN would be placed in the middle.

3.5 Policy Enforcement in SDN

Enforcing policies to the network emerges as a potential solution to solve some of the previously stated challenges regarding OMNs. Especially, considering the requirements of high priority traffic. By introducing policies it is possible to specify them to match with individual flows to perform a function. This section is a review of approaches to policy enforcement in SDN.

This thesis defines a separation between policies and flow rules. The distinction between a static policy and a flow table entry is not significant, because both parts are based on similar principles by using match conditions and corresponding actions. The main difference is that policies are defined at user-level to express the network behavior on a high-level basis. Flow rules, on the other hand, are how the actual forwarding is enforced on forwarding-level in an SDN environment. Flow rules are the standardized flow table entries of which the controller uses OF to instruct the switch how to forward the traffic. The flow table entries are straightforward, by only performing simple actions by matching with the rule, while a user-defined policy can be more complex and involve a set of measures of which should be conducted. To summarize; the user states the desired flow behavior in a policy, while the flow rule forwards the flows. The key element in performing policy enforcement would be to link policies and flow rules together, so the policy instructions are executed by the flow rules.

The administrator may define a policy which states that UDP traffic should use a particular path in the network, while the flow rules enforce this on the SDN forwarding plane by instructing the switch where to send the UDP traffic. While the administrator defines the behavior in a policy; the flow rules must be present at each switch along the path. One policy will, thus, likely be converted to multiple flow rules. Figure 3.2 illustrates the process where a user defines a policy which is compiled and dispatched into three flow rules along route X. The distinction between policies and flow rules becomes clearer when introducing dynamic policies, because this would require the controller to modify flow rules actively to cope with the changes due to them being static. Instead, if the policy states that UDP traffic should always be using the least trafficked path; then the controller would need to monitor the paths and modify/install flow rules on the switches accordingly. The dynamic policy trigger would then be located within the controller.

How policies are created and used within an SDN environment is up to the developers of the SDN application, and there are numbers of possible approaches. The essence

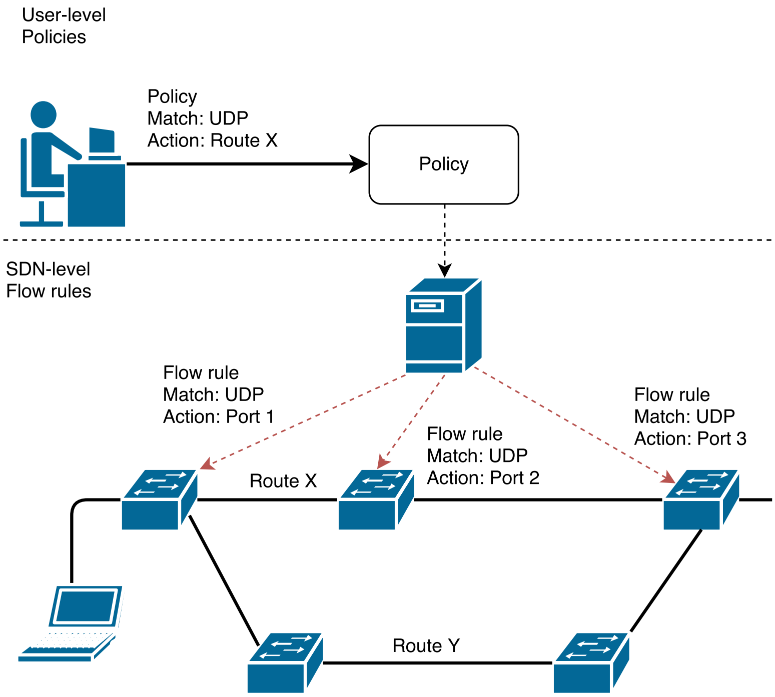


Figure 3.2: Policy vs Flow Rules

is that SDN gives the developers a set of tools to work with so they can design their customized environment. For this reason, there is really no definition on how to execute policy enforcement. Nevertheless, to specify policies, it emerges as necessary to have an interface where the administrators can specify policies that will be executed in the network somehow. Examples of some approaches are:

- User-interface for specifying flow rules directly to the switches on SDN forwarding level. For example a Graphical User Interface (GUI) or a policy list where the administrators can directly create and define flow rules while the environment is running or at boot-up. This approach limits the policies to the capabilities of which the flow rules offers, and would in practice work similar to static routing, but with with a larger set of match conditions. There is no distinction between a policy and a flow rule using this approach, and all policies would be of a static character. Regarding Figure 3.2; by using this method the user would have to define all three flow rules manually.

- User-interface to a custom developed policy environment within the application which translates the policies into the flow- and group rules. The administrators develop their policy structure/syntax and the functionality to compile them

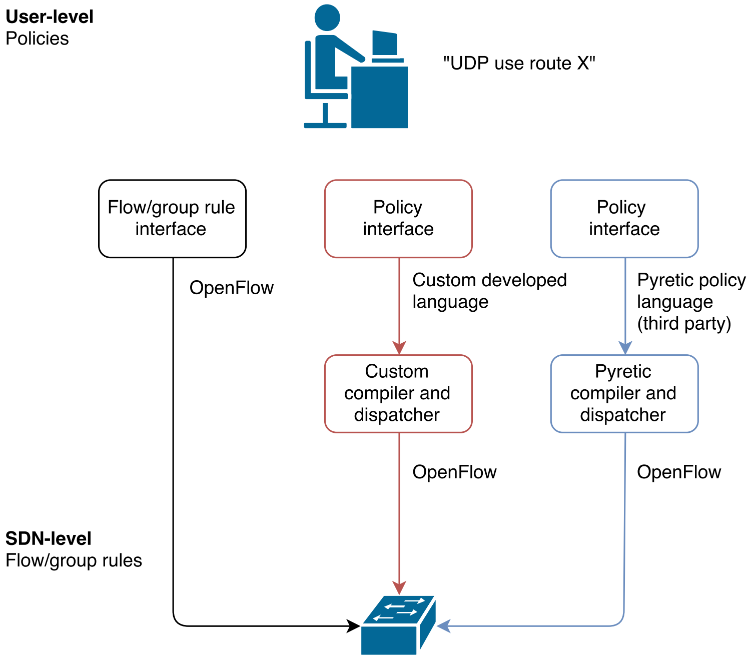

User-level
Policies
"UDP use route X"
Flow/group rule interface
Policy interface
Policy interface
OpenFlow
Custom developed language
Pyretic policy language (third party)
Custom compiler and dispatcher
Pyretic compiler and dispatcher
OpenFlow
OpenFlow
SDN-level
Flow/group rules

Figure 3.3: Three approaches to policy enforcement in SDN

into OF flow rules, and install these rules to the switches. By using this approach, the application would be more involved in the process by doing the work towards the SDN forwarding plane, while the user only works on the policy interface offered by the application. This approach would work as shown in Figure 3.2; the user defines a policy and the controller handles the rest.

– Similar to the previous custom approach, but to use third-party modules designed for policy enforcement instead of creating the environment from scratch. An example is Pyretic [28], which can be included in the applications code. Pyretic includes policy features (such as a policy syntax), and Pyretic can both convey policies into flow rules and also dispatch the rules to the nodes in the network.

Figure 3.3 illustrates the different approaches. In the figure, the SDN application provides an interface for the administrator to specify policies. The end-result on SDN forwarding-level basis will be flow and group rules, but how the actual policy environment is designed and implemented will vary. When using a flow rule interface, the flow rule becomes the policy, and the administrator must define individual flow rules to every switch along the path. The two other approaches work similarly by having a separate policy structure/language and uses the SDN application to compile the policies into flow rules and to dispatch these rules to the network.

Pyretic is an example of a platform that offers standardized ways of creating policies in SDN. Pyretic is a programming platform, which can be included in the SDN applications code, of which enables the creation of modular software such as to make sophisticated policies. Pyretic encourages programmers to focus on how to specify a network policy at a high level of abstraction, rather than how to implement it using low-level OF mechanisms, of which can be a complicated job in a large-scale network with many policies. Pyretic simplifies the policy management by compiling the policies into OF flow rules, and by dispatching the rules to the network, while the administrators only care about the policies on a high-level scale. [28]

In practice, the controller works like a white-list firewall, where only the traffic that corresponds to the flow rules can traverse through the network. Otherwise, the packet will be forwarded to the controller for processing, of which makes the controller the ideal place to perform policy enforcement. When a packet arrives the controller, it would appear natural to enforce policy checks against the packet to find possible matches, of which the application will pay attention to when generating a path. The controller can push rules to the switches both in proactive and reactive manners. Once the rules are installed on the switches, then the incoming packets will be checked against these rules and forwarded accordingly. To realize dynamic policy enforcement, the controller can modify the static flow rules accordingly by using dynamic inputs from monitoring the network. OF messages can be used to request statistics from the switches, such as flow table information to obtain information about the flows. OF is based on a request-response exchange procedure between the controller and the switches and, as a consequence to this, is this an exchange which has to be done regularly to receive the latest updates.

Chapter 4 Design

This chapter looks into possible designs for the SDN implementation regarding the associated requirements and challenges of which the Operational Military Networks are facing.

4.1 Background

The goal with the laboratory implementation is to look at how traffic engineering and policing can be solved using capabilities from the SDN suite within a dynamic military network environment. A general design specifies what is desired to perform in the laboratory implementation.

Overall, based on this thesis scope and previous discussion, emerges the Deployable Tactical Network as a network which could benefit greatly from the capabilities which SDN has to offer. The DTN is characterized as rather static, but because of its remote deployment and its purpose it has high requirements for traffic, and is also exposed to limiting factors and changes, thus underlines the dynamic factor. The need for adjusting the traffic and utilizing the resources is of great interest. The laboratory implementation is, thus, narrowed down to a primary focus on a military network which share similar characteristics with a DTN. The argument for this approach is that SDN in core networks, such as networks similar to SMCNs are already well documented and tested. The MTNs appears to require a great deal of functionality to handle the management in extreme conditions, and emerges as a too extensive topic for the given time period for this thesis. That said, the decided implementation would also be relevant for the MTN and the SMCN, due to the distinction in requirements and critical networks properties is not significant.

An assortment of important network properties is presented in section 4.2. The central elements of the design are the topology, resources, and traffic whereby it is of interest to link these features to SDN.

4.2 Network Properties Resolved by the Use of SDN

There are several scenarios and network properties which could be of interest to examine by implementing SDN. An assortment is presented in this section, where all features and properties relate to Operational Military Networks.

4.2.1 Flow Priority

The transmitted traffic within OMNs may have different priorities. The prioritized traffic should always be able to reach the destination at the other end if the connection is alive. Due to possible network capacity limitations, there must be trade-offs when the need for capacity is higher than what the network can offer. Prioritizing of traffic is, therefore, a mitigation strategy to ensure that the highest priority traffic is sent and received across the network and that the lower priority traffic has to give way.

An SDN controller can be programmed to handle traffic differently and assign priority to the flows. This can be realized either by using the priority field in the flow rules or by maintaining state about the policy priority. The advantage of using flow rules is the flexible way of forwarding packets based on various parameters from the packet, for instance, to filter out UDP traffic.

4.2.2 Flow Requirements

Particular flows may need certain requirements, such as bandwidth, delay, minimum error rate and so on. If a flow carries an important video stream, then it is of interest to forward the traffic down a path which supports the capacity requirements. To ensure that the flow maintains its requirements in the future, reservations would be necessary to prohibit other traffic streams to occupy resources on the same route (QoS).

Since the SDN controller is the Policy Decision Point, it can be developed to have full control of the incoming flows to the network. By programming the controller to obtain topology knowledge, it is possible to take forwarding decisions based on its knowledge about the flow and map this to the topology. Knowledge about particular flows must be predefined, such for example the flows minimum bandwidth requirements. The controller would also need store flow reservations in a database and maintain state.

4.2.3 Network Utilization

The network or parts of the network could fail if it is too heavy loaded regarding traffic. Traffic engineering mechanisms which can police the traffic should be implemented to avoid such scenarios. It is also advantageously that the traffic is spread across the network to prevent congestions on individual links. Regular flushing of the network could be beneficial to free resources and ensure a certain dynamic to the network.

The OF protocol has functionality which can be used to utilize the networks. For example by using group rules, the traffic can be split to several paths. Flow rules have timers which can be used to remove unused flow rules. Particular OF switches support traffic policing features which the controller can use to enforce traffic policing.

4.2.4 Smart Forwarding

A military network should be able to cope with changes, such as to detect and to make adjustments when connections break down. It is of high importance to try to find new paths to the destination if a link goes down. Re-routing is an essential ability which the network must perform in a prompt manner. Another interesting element to routing is to enforce randomness to avoid predictable flowing paths, which could be a mitigation strategy against potential eavesdroppers.

By obtaining topology knowledge, the controller can be programmed to discover alternative ways when links go down, as well use random generation algorithms to choose paths when incoming flows arrive the network. The controller could be developed to maintain state about previously chosen paths.

4.3 Chosen Implementation Design

Policy enforcement emerges as a good solution to implement the desired features presented in this chapter. This would let the administrators state desired behavior in policies as an approach to traffic engineering. To take decisions about particular flows, knowledge about flows on beforehand is necessary, such as importance and individual needs and properties. The common factor for the previously presented properties is the dynamic factor such as changes and occurring network events, of which requires monitoring mechanisms to make adjustments. The received topology information can be used together with dynamic policies to ensure adjustments of the network according to desired behavior.

Based on the earlier presented approaches to policy enforcement, it emerges as the custom policy approach is a beneficial approach because of the flexibility it brings. The approach will also give the developer more low-level control because of the ability to design the platform from scratch. For management purposes, it would be easier to define a single policy and to let the application do the compiling and the dispatching of the flow rules, instead of using a flow rule interface where all flow rules manually must be defined for each switch along the path. For this reason, a custom policy enforcement platform will be developed without including third-party policy automation to the code.

The chosen simulated network will have characteristics similar to the DTN, and the generic requirements of military networks. Following this, the design is limited to:

- Use of a single SDN controller
- Stable management links
- Variable traffic links
- Bandwidth limitations
- Unicast traffic

The application will be developed to perform dynamic policy enforcement using a reactive approach. Due to each flow may have different needs and properties, the controller should build up the forwarding gradually until the network capacity is filled up. When the network capacity is exceeding, the controller must initiate countermeasures, such as to prioritize the most relevant traffic as well as execute traffic policing to prevent congestion.

4.3.1 Objectives

The objective will be to develop an SDN dynamic policy enforcement platform capable of performing traffic engineering based on predefined policies and monitoring, in a simulated military network environment. Table 4.1 presents the summarization of the main requirements for the implementation.

Table 4.1: Design requirements

Requirement	**Description**
Flow priority	The platform should support prioritizing of individual flows, which are predefined by the administrators.
Resource reservation	Flows which may have strict requirements for resources should be offered resource guarantees (QoS).
Resource utilization	The network environment should always strive to utilize the resources in the best possible manner, based on traffic and network capacity.
Randomness	The flows should be offered unpredictable paths to mitigate potential eavesdropping and to ensure a certain dynamic to the network.
Re-forwarding	When topology changes occur, the network should re-forward the influenced flows.
Monitoring	The controller should be able to obtain topology and traffic information from the forwarding nodes.
Traffic policing	When necessary, traffic policing must be conducted to control the network and the flows.
Topology Abstraction	To ease the flow and resource management, topology abstractions should be implemented.
Management	The platform should support sound ways of managing resources and traffic for the administrators.

Chapter 5 Implementation

This chapter presents the practical laboratory work. Essential parts of the developed SDN application are described to give the reader an understanding of how the platform operates. This chapter begins with an introduction of the necessary information regarding the technology and features used in the implementation.

5.1 Introduction

5.1.1 Choice of SDN Framework

It is essential to choose the controller best to conduct the application. There are various open-source SDN controllers on the Internet. Table 5.1 shows the alternative validated open-source controllers.

An essential requirement is that the SDN framework must support the features and network properties which are to be evaluated in the implementation. To conduct traffic loading, the controller must support OF version 1.1 or higher because they include group tables. The legacy OF version does not have group tables or the support for multiple flow tables on one switch. It would be preferable to use a framework which supports the newest version as possible.

After validating the different controllers, Ryu was chosen as the framework for the laboratory implementation. Some of the validated controllers have not been updated in years, while Ryu is regularly updated. Ryu supports all versions of OF. Thus, the necessary features to implement a traffic engineering platform in SDN. An advantage with Ryu is that the controller has many existing modules which can be used together with customized applications. Ryu is also well documented, and there exist many examples on the Internet which can contribute to simplifying the development process.

Table 5.1: Open-source SDN frameworks

Controller	Description
NOX	The first OF controller developed, used as basis for many subsequent implementations [17]. Based on C++ language.
POX	Python version of NOX. Popular controller among academics to explore SDN.
OpenDayLight	Java based SDN framework with support for OF. OpenDayLight is a collaboration project with members from big enterprises such as Cisco, Juniper and Citrix [26].
Beacon	A high performance OF controller developed in 2010. Widely used for teaching and research in academia [10]. Written in Java.
Ryu	A Python based SDN controller framework with OF support and with a variety of software components, making it easy for developers to create new network management applications [9].
FloodLight	An open community SDN project. The framework is based on the Beacon controller and a large support community with contributors to the development project. Written in Java. [12]
ONOS	A new upcoming SDN platform developed by a community excising of 50 partners and collaborators. Written in Java. [25]

5.1.2 Ryu

Ryu is a component based SDN framework, developed in Python. Ryu is the Japanese word for *flow*. Ryu provides software components with well-defined interfaces which make it easy for developers to create new network management and control applications. One of the strengths of Ryu is that it supports multiple southbound protocols for managing network devices, and Ryu supports all the released versions of OF, where version 1.5 is the latest version. The framework includes a controller, of which can run customized Ryu applications. An advantage of the Ryu framework is that it contains modules which can be included in customized code, avoiding the need of developing an application from scratch. [9]

Figure 5.1 presents the Ryu framework and its main elements. Ryu supports many libraries, ranging from support for multiple southbound protocols to various packet processing operations. The OF protocol is one of the supported southbound protocols, and the framework also includes an internal OF controller. The controller is the entity which manages the switches and events. The *Ryu-manager* is the main

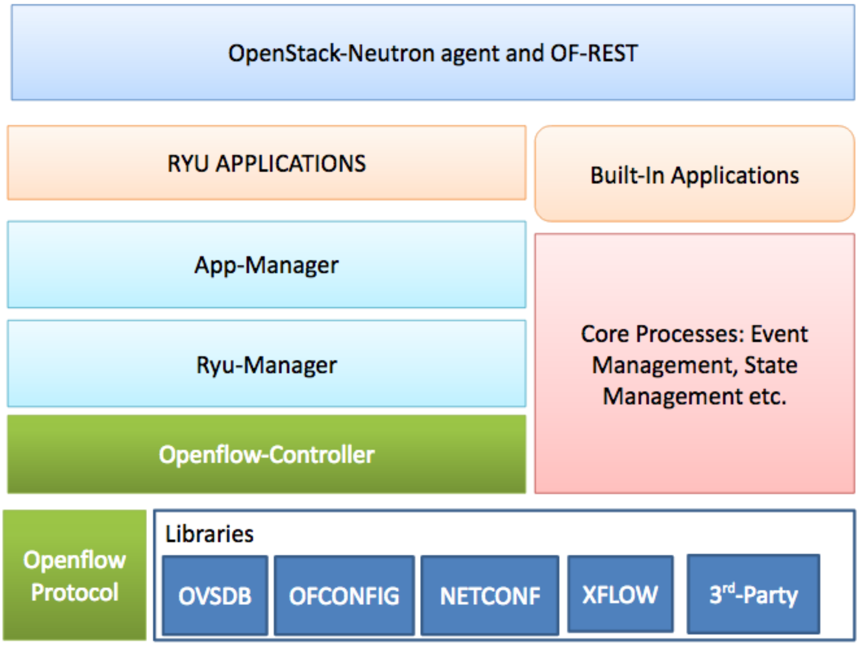


Figure 5.1: Ryu architecture [24]

executable, and when running, it creates a listener in which can connect to the OF switches. The *App-manager* is the essential component for all Ryu applications, due to all applications inherit functionality from the App-manager's RyuApp class. The *core-process* component in the architecture includes event management, messaging, in-memory state management, etc. The top layer illustrated in the architecture is the northbound Application Programming Interface (API), where supported plug-ins can communicate with Ryu's OF operations. [24]

The essential element is the *Ryu applications*, which is where the control logic and behavior is defined. The Ryu application bases its operations on event handlers, where the network events are passed up to be processed by the application. Code sequence 5.1 gives an example of such an event listener. In this snipped it is assumed that functionality inherits from the *App-manager*. By importing two libraries included with the Ryu framework, event and packet processing is possible. The event listener will be triggered every time an OF switch forwards a packet to the controller[1]. The administrator develops the packet processing behavior.

[1] Which is every time an incoming flow does not match with any rule.

Code sequence 5.1 Event listener for incoming packets in Ryu syntax.

```
from ryu.controller import ofp_event
from ryu.lib.packet import packet

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        pkt = packet.Packet(msg.data)
        #Do handling of packet
```

5.1.3 Mininet

Mininet is a network emulator based on Python, which runs on a regular computer, used to simulate physical network behavior. A Mininet host behaves and operates as a real computer, making it possible to customize and run regular programs[2]. Packets sent to the network are processed by virtual Ethernet interfaces, but operates in the same way as physical interfaces, making it ideal for network simulation. [36]

Mininet supports both legacy switches and an assortment OF switches and provides the opportunity to link external OF controllers to the emulated network, such as the Ryu controller. Mininet can be used to simulate different network topologies or network scenarios. A link can be configured with various properties such as speed, jitter, delay, and loss. It is possible to simulate changing network environments by including Python loops and conditions in the topology script.

Mininet has a GUI called MiniEdit which can be used to design visual network topologies [36]. It also has a Command Line Interface (CLI) directly towards the network nodes, which can be used to execute commands, apply rules or show information about the network nodes, while the network emulation is running. The commands can, for example, be used to generate traffic in the network or dump flow table information from the OF switches. As of today, the latest supported OF version in Mininet is 1.3.

5.1.4 Open vSwitch

One of the supported OF switches in Mininet is the Open vSwitch (OVS). OVS is a well known open-source implementation of a distributed virtual multilayer switch. It is designed to enable network automation through programmatic extension, while still supporting standard management interfaces and protocols such as OF.

[2] For example Wireshark to perform traffic analysis

Table 5.2: Software versions used in the implementation.

Software	**Description**
Ryu 4.1	SDN framework.
VirtualBox 5.0.2	Virtual software used to run Mininet.
Mininet 2.2.1	Network topology environment.
Open vSwitch 2.5	Virtual multilayer switch with OF support.
Python 2.7.6	Programming language.
Pycharm 5.0.4	Python Integrated Development Environment (IDE).
Linux Mint 17.3	Host OS
Ubuntu 14.04	Virtual Machine 1

In an OVS implementation, a database server and a switch daemon[3] are used. To manage these components, OVS introduces Open vSwitch Database Management Protocol (OVSDB), which manages logical OF datapaths[4]. Controllers use OF to install flow state in switches, while OVSDB manages the switch itself. An OVS instance can support multiple logical bridges, where there is at least one OF controller for each bridge. [27]

The OVSDB management interface is used to perform management and configuration operations on the OVS instance. Examples of usage are to manage tables, queues and statistics. OVS has various tools included which is capable of doing several of the same functions as the OF protocol, for example, to clean flow tables. However, OVSDB does not perform per-flow operations, leaving those instead to OF.

5.2 The Laboratory Implementation

The laboratory implementation is structured as illustrated in Figure 5.2. The Ryu framework is designed to run on the host computer with a TCP connection to the emulated Mininet network topology[5]. The controller and the network are placed on different hosts to make the simulation most realistic. The figure illustrates where the various technology is positioned to conduct the lab, whereby Table 5.2 lists the details about software versions.

[3] A daemon that manages and controls any number of OVS switches on the local machine.

[4] *Datapath* and *Bridge* are another terms for a switch.

[5] The figure is an overview of a Mininet network. The topology used in the implementation is described later.

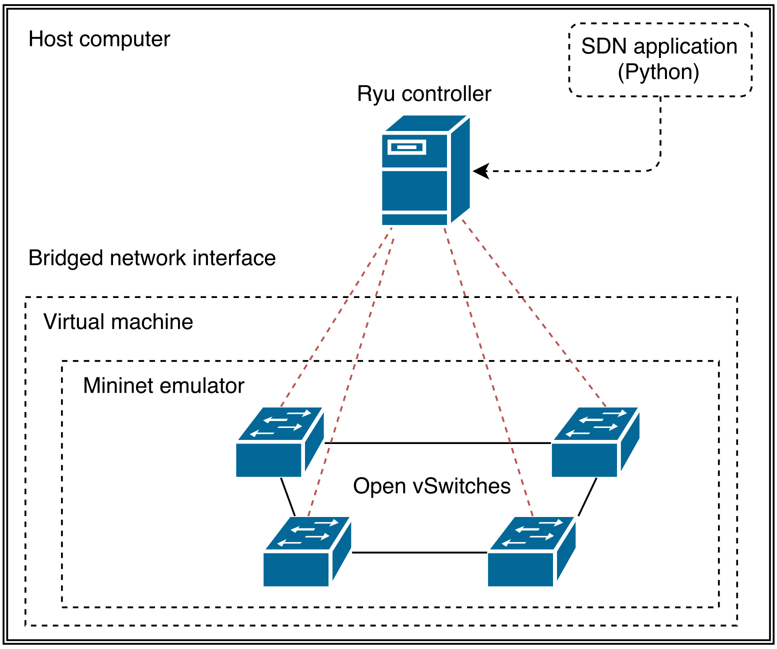


Figure 5.2: Testbed structure

5.2.1 Network Emulation in Mininet

The chosen network emulator was Mininet, due to it emerges as a robust platform to simulate customized military network environments. Following is a short overview of the most important commands used in the testbed implementation.

To run a customized Mininet topology:

```
$  sudo mn --custom military_network_topology.py
```

Within the customized topology, the necessary settings must be defined. This includes creation and configuring of nodes with IP addresses, hostnames, and Data Path Identifier (DPID) addresses[6]. In this case, will Mininet create an OVS switch.

```
s1 = net.addSwitch('s1', cls = OVSKernelSwitch, dpid = '1')
h1 = net.addHost('h1', cls = Host, ip = '10.10.10.101', defaultRoute = None)
```

Defining and configuring of links between nodes with various network properties:

[6] DPIDs are used to identify switches.

```
net.addLink(h1, s1, bw = 10, delay = '5ms', loss = 5)
```

The IP address of the controller must be configured to establish the connection with the switches. Since the controller is running separated from the Mininet Virtual Machine (VM), it must be configured as remote:

```
c0 = net.addController(name = 'c0', controller = RemoteController,
ip = '129.241.208.193',protocol = 'tcp', port = 6633)
```

Mininet CLI

The Mininet CLI is used to run commands on the OVS switches, and is a necessary feature for diagnostics. Two tools that are useful is *ovs-vsctl*, which is a high-level interfaces to the OVS database, and *ovs-ofctl* which is used for switch administration.

To configure a switch (s1) with OF 1.3:

```
ovs-vsctl set bridge s1 protocols = OpenFlow13
```

Dump flow and group tables for a switch:

```
sudo ovs-ofctl -O OpenFlow13 dump-flows s1
sudo ovs-ofctl -O OpenFlow13 dump-groups s1
```

5.2.2 Ryu

As earlier presented, Ryu's primary executable is the Ryu-manager and is used to run the applications. The following command starts the application script:

```
$  ./ryu-manager sdn_app.py --observe-links
```

–observe-links is added to the command to observe the link events as output from Ryu-manager. By issuing the start-up command, the controller will become visible for the switches, of which will lead to the discovery procedure and the establishment of the OF channel, presented in Section 2.2.

5.3 Policy Enforcement Application

This section will present important information about the application, to give the users of the application a better understanding of the essential operations and how to use the application. The source code is published together with this thesis.

The policy enforcement platform provides a simplified policy list[7], where policies of which should be applied to flows are specified. Each policy exists of match conditions, actions and a priority. The policy list loads at start-up of the controller, and incoming flows will be checked against this list. Flows will only run this check when forwarded to the controller, thus is when no flow rules exist on the switches[8].

Below is an example of the policy structure. For this example, the policy will be executed when the hosts source IP address is equal to the address specified in the policy.

```
policy001 = policy_manager.Policy()
policy001.match(ip_src="10.10.10.102")
policy001.action(bandwidth_requirement=5)
policy001.priority(4)
```

Match conditions are defined to map a policy with a particular flow. The policy requires at least one match condition to work. Parameters from the packet are tested against the policy list, and the application will fetch the policies which match with the packet's parameters. The policy list is applied to one direction of the flow, meaning that when a host 1 (h1) initiates a connection to host 2 (h2), the policy-check will run twice, one time for h1-h2 and one for h2-h1. This provides the opportunity to enforce different policies for each direction of the flow: the packets from h1 to h2 can travel on a different path than the response from h2 to h1. Table 5.3 lists the optional policy match conditions.

The policies are defined to do actions to the incoming flows. Table 5.4 lists possible policy actions. A policy can be configured to have many actions, which ensures customized of flows on a granular level. On the other hand, there are also conflicting actions, such as including blocking and bandwidth requirements to the same policy. Nevertheless, this should be rather obvious by the administrator.

Each policy is configured to have priority. The priority defines which policy is most important. The priority ranges from 1 to 20, where 1 is the highest priority. When a new incoming flow arrives, the policy enforcement application will find the matching

[7] The file: policy_inputs.py

[8] This is at first-entry of the flow, or after a flow timeout or flow deletion.

Table 5.3: Policy match conditions

Match condition	**Values**	**Descripion**
ip_dst	IP address (string)	Destination IP address
ip_src	IP address (string)	Source IP address
protocol	Protocol number (integer)	The packets protocol
eth_dst	Ethernet address (string)	Destination MAC address
eth_src	Ethernet address (string)	Source MAC address
eth_type	Ethernet type number (integer)	The Ethernet type

policies and sort them according to the priority. The top priority policies will be enforced first, and may lead to the lower policies not being enforced. Instead, the administrator should include multiple actions in one policy instead of many policies for the same flow with one action each. If a policy does not have any priority predefined, the application will construct a priority based on the number of match conditions the policy possesses. This indicates that the most specific policies will get the highest priority if the administrator does not define any priority.

Flow priority is the primary factor for the application, and it is of particular importance when the network is starting to reach its maximum capability limits. When reaching the congestion point, the application will have to make adjustments, both taking the network resources and the traffic in consideration. Low priority flows will need to give away to high priority traffic, and this priority is rooted in the policy priorities[9].

[9] The flow gets its priority from the policy which it has a match with.

Table 5.4: Policy actions.

Action	**Values**	**Description**
bandwidth_requirement	1-9999X	The minimum required bandwidth for the flow in Mbit/s. The specified requirement will be reserved by the application for the particular flow throughout the period where the flow is active. The requirement is strict, meaning an absolute requirement (QoS).
bandwidth_requirement _nonstrict	True/False	This property should be applied to the policy if the bandwidth requirement is preferring (and not the absolute minimum). Non-strict policies can handle lower bandwidths, but will strive to achieve the requirement.
allow_load _balance	True/False	This setting will allow or deny the particular flow to be traffic loaded (split traffic onto multiple paths). A feature used by the flow in situations where it is necessary to split the traffic to achieve the bandwidth requirement.
random_routing	True/False	Needs to be set to *true* to choose a random flow path and to re-forward the flow randomly.
block	True/False	Needs to be set to *true* to block a particular flow.
traffic_class	1-3	Network resources are classified in separate traffic class pools, based on their bandwidth capabilities. This setting defines which pool the flow should use. Traffic class 1 is the best class, while 3 is the poorest.

5.3.1 Logical Policy Design

Figure 5.3 illustrates the logical design of the forwarding decision process when incoming flows arrive the controller. The administrators define policies in an interface, whereby a policy list stores the policies. When incoming flows reach the controller, the application will read the policy list to check packet parameters against match conditions.

When a policy is enforced and used in the network, it is copied[10] to another list called *running policies*. This list saves the enforced policies, so the controller keeps the state of every running policy on each path in the network. The forwarding decision process will also inspect this list if it is necessary to remove a policy to make room for a higher priority policy. Despite this, it is important to note that the figure only displays the policy process; the controllers monitored view of the topology and traffic influences the forwarding decision process.

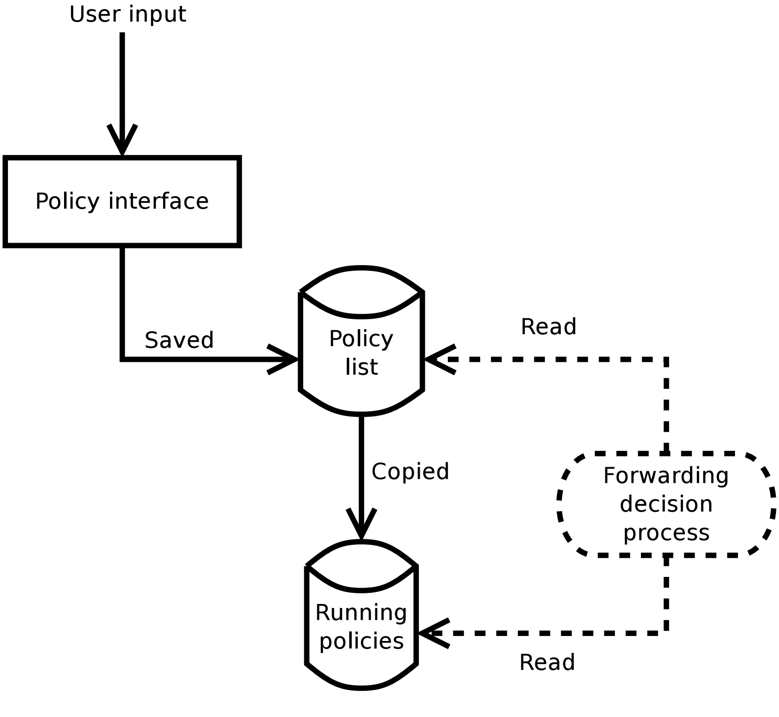


Figure 5.3: Policy storage

[10] It is copied, but additional parameters are added, such as chosen path.

5.3.2 Policy Matching

Technically, all policies are created as individual objects, by which a policy contains match conditions, actions and a priority. When a new flow arrives at the controller, the various parameters (ref. Table 5.3) will be extracted from the packet and tested according to the procedure illustrated in Figure 5.4[11]. Since multiple policies may match with an incoming flow, it is necessary to separate them with priorities to be able to decide which policy to enforce. After fetching the matching policies, a list saves them and sorts policies by priority.

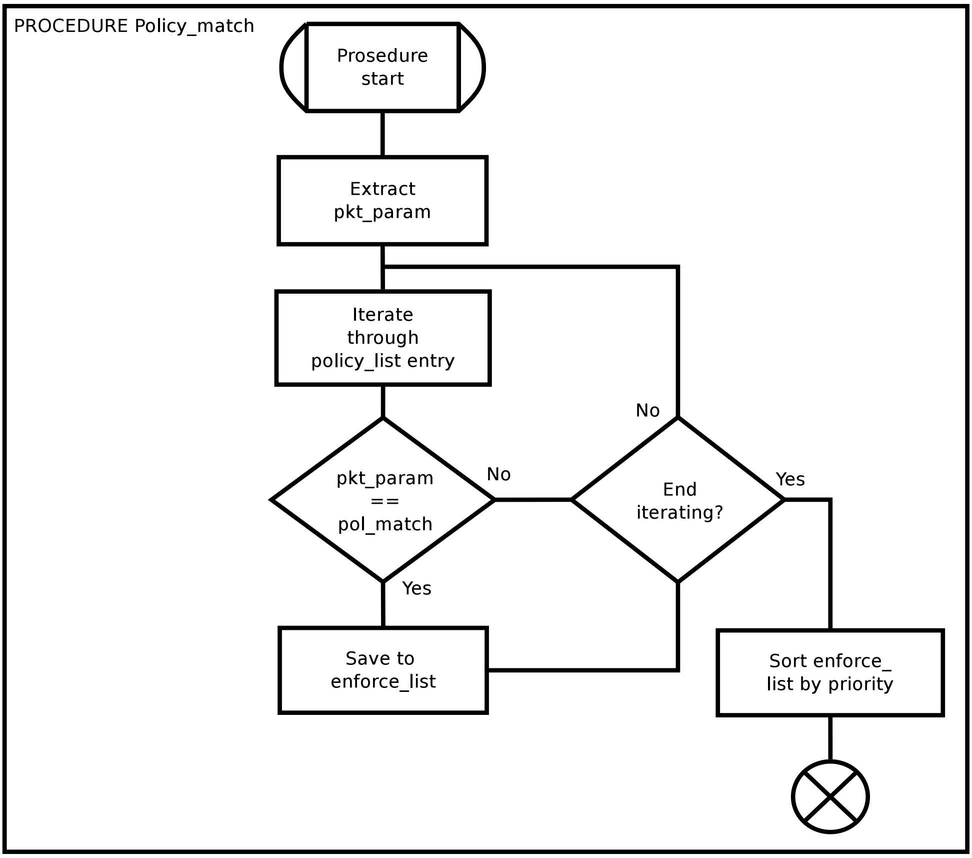


Figure 5.4: Matching and sorting policies

[11] The figure is drawn in Specification and Description Language (SDL) [4], where the symbols are explained in Appendix B.

5.3.3 Policy Enforcement Design

Figure 5.5 displays the overall policy enforcement process. The process is composed of several procedures which serve different purposes. The figure shows the overall process and the procedure calls, of which is calling the external procedures, such as the *policy_match* procedure which is earlier illustrated in Figure 5.4.

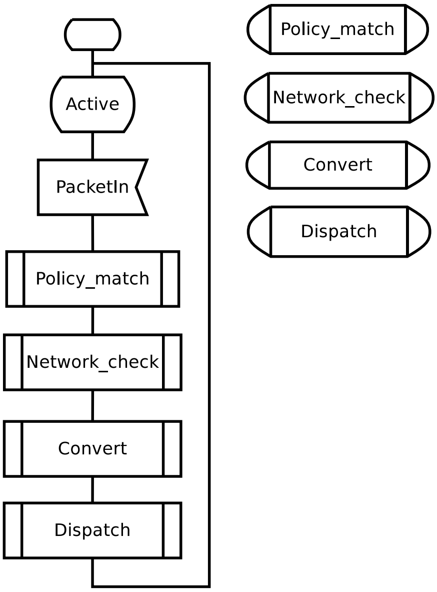


Figure 5.5: Policy enforcement process

When a packet arrives the controller, the application will perform the policy matching procedure (*policy_match*) as earlier described. The controller will start enforcing the policy with the highest priority first, and it will continue down in the list until the network accepts the policy, thus being the *network_check* procedure. If the highest policy is approved, then only this policy will be enforced. If it is not accepted then the next policy in line will be enforced, and so on.

Once the policy is accepted, the application will then compile the policy into flow rules, where it will configure the MAC source and destination addresses of the communicating entities as match conditions for the flow rules. The flow rules are always compiled in this manner, independent of the match condition stated in the policy, and it makes a flow rule unique for each flow direction for a particular pair of nodes. The application uses obtained topology knowledge to locate the nodes and to compute a suitable path, whereby the controller iterates over the switches along the path and dispatches/installs the corresponding flow rules.

Chapter 6
Validation

This chapter is about the testing of the developed policy enforcement application, to see the enforcement of policies and the utilizing of resources when changes and events occur. This section is presented as a chain of activities using a single topology, but where policies are defined based on military assumptions and requirements.

6.1 Details

The main objective of the testing of the different scenarios is to see how the policy enforcement application operates and how it utilizes dynamic policies in changing conditions. Table 6.1 lists the commands used to generate traffic and conduct analysis in the testbed.

Table 6.1: Tools used

Tool	**Description**
ping	Measure liveness and delay
iperf	Used to generate traffic and do measurements
ovs-ofctl	Dump flow and group table information.
tcpdump	Traffic analysis

6.1.1 Network Topology

The Mininet network environment used in this validation is illustrated in Figure 6.1, and the Mininet script is attached in Appendix A. All nodes within the network share the same properties and capabilities, but the links vary in bandwidth capacity, as listed in Table 6.4. The same topology will be used throughout the validation phase, but links may be disconnected. When this is the case, then it is stated in the introduction to the scenario.

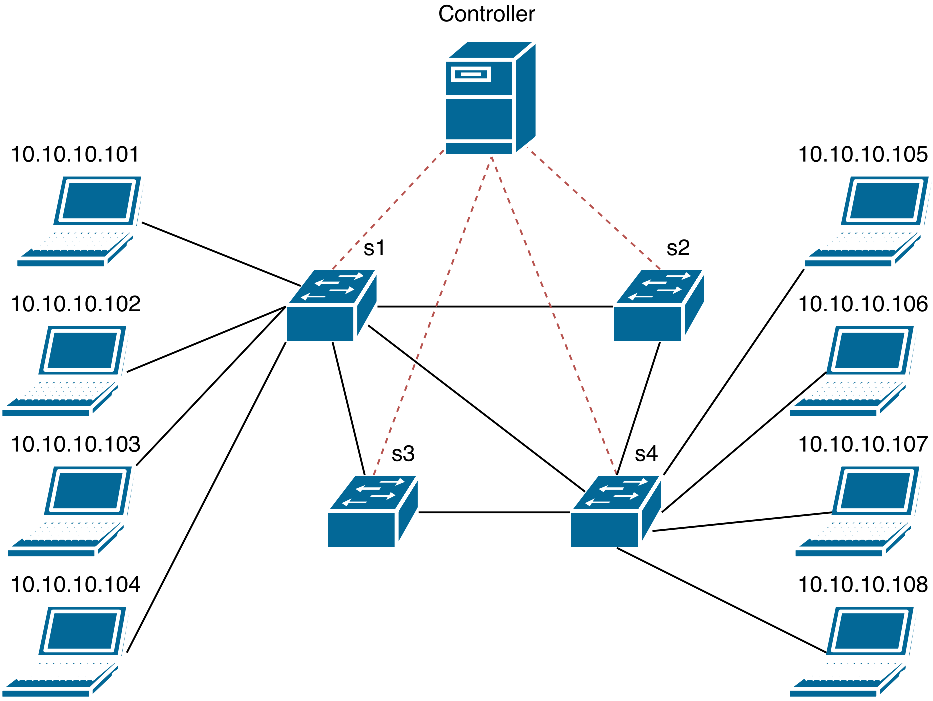


Figure 6.1: Mininet topology used in the tests

The network topology used represents a simplified network where several clients are connected. A client does not necessary need to be a computer, but rather a device with an IP address, such as example a tunnel interface or a traffic encryption device. The essence is, however, to forward the traffic generated from the end-nodes within the network using SDN capabilities. Table 6.2 and 6.3 list the adresses of hosts and nodes.

6.1.2 Policies Used in the Validation

The policies applied in this testbed are all defined on beforehand in the policy list. That said, each policy is presented as the scenarios go on. The essential policy properties are previously explained in Table 5.3 (match conditions) and Table 5.4 (policy actions).

Table 6.2: Hosts

Host	**IP address**
h1	10.10.10.101
h2	10.10.10.102
h3	10.10.10.103
h4	10.10.10.104
h5	10.10.10.105
h6	10.10.10.106
h7	10.10.10.107
h8	10.10.10.108

Table 6.3: Switches

Switch	**Dpid**
s1	1
s2	2
s3	3
s4	4

Table 6.4: Link speeds

Link	**Capacity**
s1 - s2	2 Mbit/s
s1 - s3	2 Mbit/s
s1 - s4	4 Mbit/s
s2 - s4	2 Mbit/s
s3 - s4	2 Mbit/s

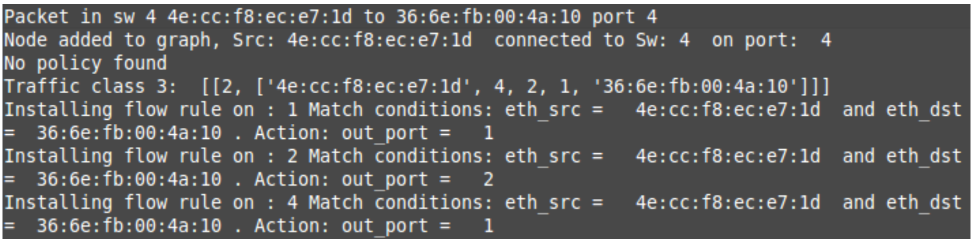
6.2 Priority and Traffic Utilization

Firstly, we define a policy for h1 because the traffic from this computer requires a bandwidth of 3 Mbit/s to achieve satisfying quality. We specify that the traffic can be split on multiple paths if necessary to achieve its bandwidth requirements. The priority is 5, which indicates that the priority is fairly high. This results in the following policy, which is added to the policy list:

```
policy001 = policy_manager.Policy()
policy001.match(ip_src="10.10.10.101")
policy001.action(bandwidth_requirement=3, allow_load_balance=True)
policy001.priority(5)
```

When pinging from h1 to h5, illustrated in Figure 6.2 and 6.3:

Explanation of Figure 6.2: h1 pings h5 for the first time, of which initiates an Address Resolution Protocol (ARP) broadcast procedure by the application to locate h5. When h5 receives this request, it will respond to h1. The application only performs policy enforcement when it has location knowledge about source and

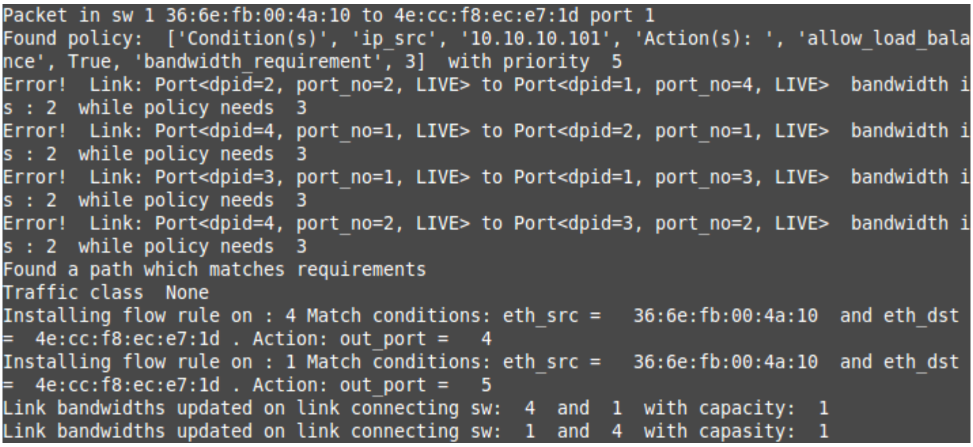


```
Packet in sw 4 4e:cc:f8:ec:e7:1d to 36:6e:fb:00:4a:10 port 4
Node added to graph, Src: 4e:cc:f8:ec:e7:1d  connected to Sw: 4  on port:  4
No policy found
Traffic class 3:  [[2, ['4e:cc:f8:ec:e7:1d', 4, 2, 1, '36:6e:fb:00:4a:10']]]
Installing flow rule on : 1 Match conditions: eth_src =   4e:cc:f8:ec:e7:1d  and eth_dst
=  36:6e:fb:00:4a:10 . Action: out_port =   1
Installing flow rule on : 2 Match conditions: eth_src =   4e:cc:f8:ec:e7:1d  and eth_dst
=  36:6e:fb:00:4a:10 . Action: out_port =   2
Installing flow rule on : 4 Match conditions: eth_src =   4e:cc:f8:ec:e7:1d  and eth_dst
=  36:6e:fb:00:4a:10 . Action: out_port =   1
```

Figure 6.2: No policy found

destination, so in practice is the response back from h5 to h1 the first flow which is handled by the policy enforcement procedure. However, the flow does not match with any policies. As a result, a standard path without any resource reservation will be generated using traffic class 3[1]. Traffic class 3 indicates the poorest capacity paths, and this is due to flows without any policy have the lowest priority in the network.

[1] The algorithm is described in Section 7.9



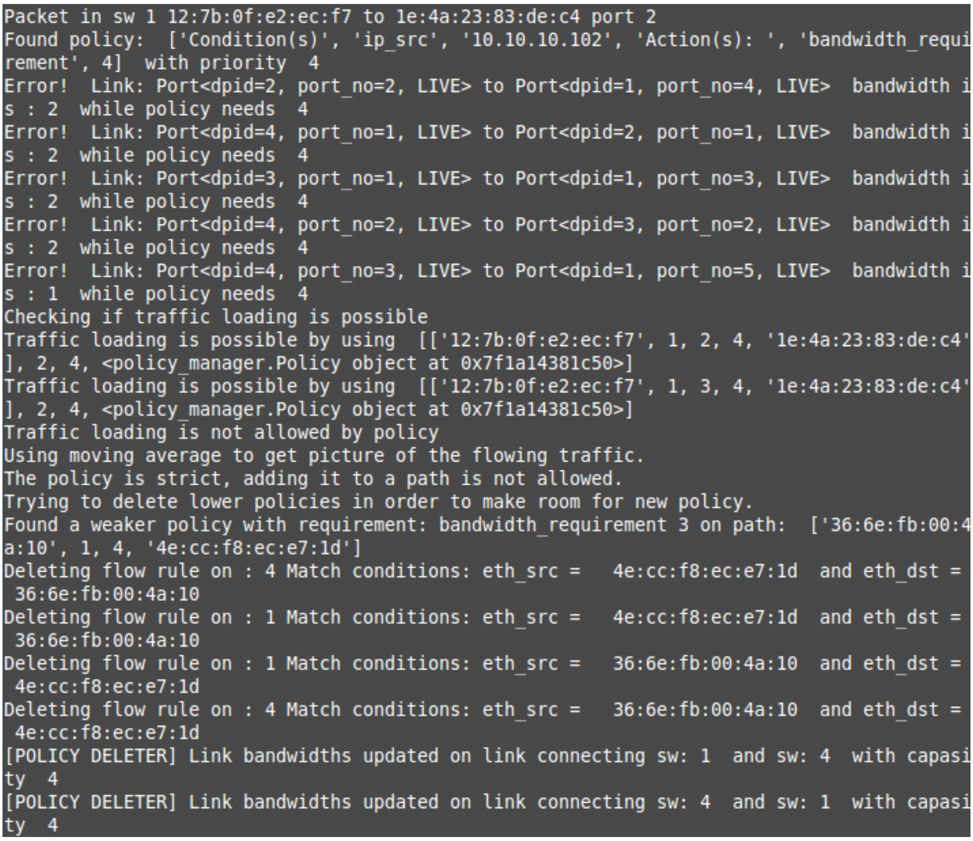
```
Packet in sw 1 36:6e:fb:00:4a:10 to 4e:cc:f8:ec:e7:1d port 1
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'allow_load_bala
nce', True, 'bandwidth_requirement', 3]  with priority  5
Error!  Link: Port<dpid=2, port_no=2, LIVE> to Port<dpid=1, port_no=4, LIVE>  bandwidth i
s : 2  while policy needs  3
Error!  Link: Port<dpid=4, port_no=1, LIVE> to Port<dpid=2, port_no=1, LIVE>  bandwidth i
s : 2  while policy needs  3
Error!  Link: Port<dpid=3, port_no=1, LIVE> to Port<dpid=1, port_no=3, LIVE>  bandwidth i
s : 2  while policy needs  3
Error!  Link: Port<dpid=4, port_no=2, LIVE> to Port<dpid=3, port_no=2, LIVE>  bandwidth i
s : 2  while policy needs  3
Found a path which matches requirements
Traffic class  None
Installing flow rule on : 4 Match conditions: eth_src =   36:6e:fb:00:4a:10  and eth_dst
=  4e:cc:f8:ec:e7:1d . Action: out_port =   4
Installing flow rule on : 1 Match conditions: eth_src =   36:6e:fb:00:4a:10  and eth_dst
=  4e:cc:f8:ec:e7:1d . Action: out_port =   5
Link bandwidths updated on link connecting sw:  4  and  1  with capacity:  1
Link bandwidths updated on link connecting sw:  1  and  4  with capasity:  1
```

Figure 6.3: Policy found

Explanation of Figure 6.3: After the response from h5, then h1 knows the MAC to h5, and will start transmitting regular ping requests. The application handles the first ping-request and finds a match with the policy. The controller ends up installing this direction of the flow differently from the first, due to the policy forces the flow to use a link which supports the policy bandwidth requirement. After installing the flow, the link capacity is updated, whereby only 1 Mbit/s is left for other flows to use.

We want to initiate a new connection between h2 and h6, where the bandwidth requirement and priority is higher than the previous policy. We define a new policy and start pinging from h2, illustrated in Figure 6.4.

```
policy002 = policy_manager.Policy()
policy002.match(ip_src="10.10.10.102")
policy002.action(bandwidth_requirement=4)
policy002.priority(4)
```

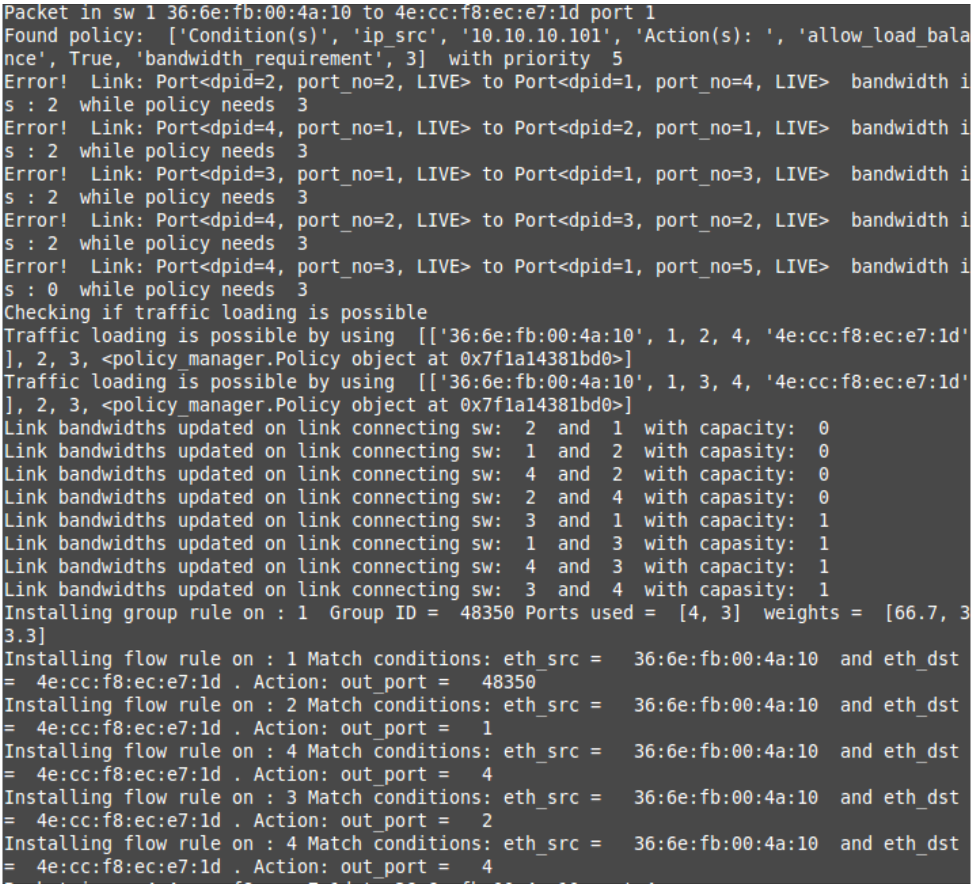


```
Packet in sw 1 12:7b:0f:e2:ec:f7 to 1e:4a:23:83:de:c4 port 2
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.102', 'Action(s): ', 'bandwidth_requi
rement', 4]  with priority  4
Error!  Link: Port<dpid=2, port_no=2, LIVE> to Port<dpid=1, port_no=4, LIVE>  bandwidth i
s : 2  while policy needs  4
Error!  Link: Port<dpid=4, port_no=1, LIVE> to Port<dpid=2, port_no=1, LIVE>  bandwidth i
s : 2  while policy needs  4
Error!  Link: Port<dpid=3, port_no=1, LIVE> to Port<dpid=1, port_no=3, LIVE>  bandwidth i
s : 2  while policy needs  4
Error!  Link: Port<dpid=4, port_no=2, LIVE> to Port<dpid=3, port_no=2, LIVE>  bandwidth i
s : 2  while policy needs  4
Error!  Link: Port<dpid=4, port_no=3, LIVE> to Port<dpid=1, port_no=5, LIVE>  bandwidth i
s : 1  while policy needs  4
Checking if traffic loading is possible
Traffic loading is possible by using  [['12:7b:0f:e2:ec:f7', 1, 2, 4, '1e:4a:23:83:de:c4'
], 2, 4, <policy_manager.Policy object at 0x7f1a14381c50>]
Traffic loading is possible by using  [['12:7b:0f:e2:ec:f7', 1, 3, 4, '1e:4a:23:83:de:c4'
], 2, 4, <policy_manager.Policy object at 0x7f1a14381c50>]
Traffic loading is not allowed by policy
Using moving average to get picture of the flowing traffic.
The policy is strict, adding it to a path is not allowed.
Trying to delete lower policies in order to make room for new policy.
Found a weaker policy with requirement: bandwidth_requirement 3 on path:  ['36:6e:fb:00:4
a:10', 1, 4, '4e:cc:f8:ec:e7:1d']
Deleting flow rule on : 4 Match conditions: eth_src =   4e:cc:f8:ec:e7:1d  and eth_dst =
 36:6e:fb:00:4a:10
Deleting flow rule on : 1 Match conditions: eth_src =   4e:cc:f8:ec:e7:1d  and eth_dst =
 36:6e:fb:00:4a:10
Deleting flow rule on : 1 Match conditions: eth_src =   36:6e:fb:00:4a:10  and eth_dst =
 4e:cc:f8:ec:e7:1d
Deleting flow rule on : 4 Match conditions: eth_src =   36:6e:fb:00:4a:10  and eth_dst =
 4e:cc:f8:ec:e7:1d
[POLICY DELETER] Link bandwidths updated on link connecting sw: 1  and sw: 4  with capasi
ty  4
[POLICY DELETER] Link bandwidths updated on link connecting sw: 4  and sw: 1  with capasi
ty  4
```

Figure 6.4: Policy deleted

Explanation of Figure 6.4: The controller matches the incoming flow with policy002, but discovers that it is unable to achieve its bandwidth requirements, and is therefore forced to look for alternatives. Policy002 is strict, meaning that the bandwidth requirement is absolute. Traffic loading of the flow is also not allowed. The controller is given no opportunities in this situation but look for policies with a lower priority, and try to remove them to free resources. The controller finds the former flow with the lower priority and deletes it so the new flow can use this path.

After enforcing policy002, we initiate a new ping from h1 to h5 (policy001):

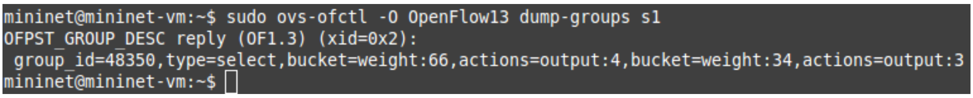


```
Packet in sw 1 36:6e:fb:00:4a:10 to 4e:cc:f8:ec:e7:1d port 1
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'allow_load_bala
nce', True, 'bandwidth_requirement', 3]  with priority  5
Error!  Link: Port<dpid=2, port_no=2, LIVE> to Port<dpid=1, port_no=4, LIVE>  bandwidth i
s : 2  while policy needs  3
Error!  Link: Port<dpid=4, port_no=1, LIVE> to Port<dpid=2, port_no=1, LIVE>  bandwidth i
s : 2  while policy needs  3
Error!  Link: Port<dpid=3, port_no=1, LIVE> to Port<dpid=1, port_no=3, LIVE>  bandwidth i
s : 2  while policy needs  3
Error!  Link: Port<dpid=4, port_no=2, LIVE> to Port<dpid=3, port_no=2, LIVE>  bandwidth i
s : 2  while policy needs  3
Error!  Link: Port<dpid=4, port_no=3, LIVE> to Port<dpid=1, port_no=5, LIVE>  bandwidth i
s : 0  while policy needs  3
Checking if traffic loading is possible
Traffic loading is possible by using  [['36:6e:fb:00:4a:10', 1, 2, 4, '4e:cc:f8:ec:e7:1d'
], 2, 3, <policy_manager.Policy object at 0x7f1a14381bd0>]
Traffic loading is possible by using  [['36:6e:fb:00:4a:10', 1, 3, 4, '4e:cc:f8:ec:e7:1d'
], 2, 3, <policy_manager.Policy object at 0x7f1a14381bd0>]
Link bandwidths updated on link connecting sw:  2  and  1  with capacity:  0
Link bandwidths updated on link connecting sw:  1  and  2  with capasity:  0
Link bandwidths updated on link connecting sw:  4  and  2  with capacity:  0
Link bandwidths updated on link connecting sw:  2  and  4  with capasity:  0
Link bandwidths updated on link connecting sw:  3  and  1  with capacity:  1
Link bandwidths updated on link connecting sw:  1  and  3  with capasity:  1
Link bandwidths updated on link connecting sw:  4  and  3  with capacity:  1
Link bandwidths updated on link connecting sw:  3  and  4  with capasity:  1
Installing group rule on : 1  Group ID =  48350 Ports used =  [4, 3]  weights =  [66.7, 3
3.3]
Installing flow rule on : 1 Match conditions: eth_src =   36:6e:fb:00:4a:10  and eth_dst
=  4e:cc:f8:ec:e7:1d . Action: out_port =   48350
Installing flow rule on : 2 Match conditions: eth_src =   36:6e:fb:00:4a:10  and eth_dst
=  4e:cc:f8:ec:e7:1d . Action: out_port =   1
Installing flow rule on : 4 Match conditions: eth_src =   36:6e:fb:00:4a:10  and eth_dst
=  4e:cc:f8:ec:e7:1d . Action: out_port =   4
Installing flow rule on : 3 Match conditions: eth_src =   36:6e:fb:00:4a:10  and eth_dst
=  4e:cc:f8:ec:e7:1d . Action: out_port =   2
Installing flow rule on : 4 Match conditions: eth_src =   36:6e:fb:00:4a:10  and eth_dst
=  4e:cc:f8:ec:e7:1d . Action: out_port =   4
```

Figure 6.5: Traffic loading applied to flow

Explanation of Figure 6.5: Due to that policy002 forced the application to delete the flow that was using policy001 means that there is there are no flow rules for h1 and h5 on the switches anymore. In other words, if h1 tries to ping h5, the packet will be forwarded to the controller for re-processing.

Since policy001 is defined to allow traffic loading, the flow is permitted to be split into two different paths. This ensures that the flow achieves its bandwidth requirement by using portions of bandwidth capacity from each path. Figure 6.6 shows the created rule on s1.



```
mininet@mininet-vm:~$ sudo ovs-ofctl -O OpenFlow13 dump-groups s1
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=48350,type=select,bucket=weight:66,actions=output:4,bucket=weight:34,actions=output:3
mininet@mininet-vm:~$ 
```

Figure 6.6: Weighted group rule

6.2.1 Summary

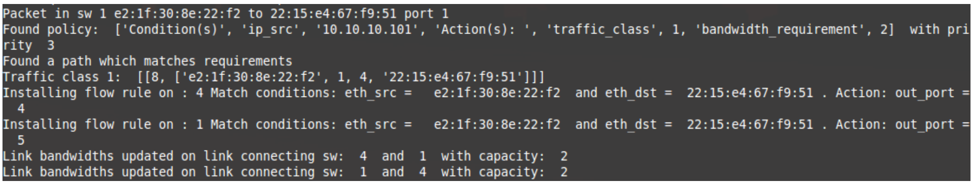
The first incoming flow (policy001) takes the path s1-s4, due to it is the only link which has the capacity of which the policy requires. The application always strives to use paths that fulfills the bandwidth requirement, and only splits traffic if necessary. When the next flow (policy002) arrives the network, the first flow will be deleted due to it having a lower priority, because the new flow is forced use path s1-s4. By the removal of policy001, the flow must be re-processed by the application when it arrives the network after deletion, of which the application finds it necessary to split the traffic into two paths to achieve the required bandwidth. The final result for policy001 (with a bandwidth requirement of 3 Mbit/s) is reserving 2 Mbit/s on path s1-s2-s4 and 1 Mbit/s on path s1-s3-s4.

6.3 Traffic Classes

We assume that the previous policies have timed out. We want to make some adjustments to existing policies. We have noticed that the minimum bandwidth for h1 is actually 2 Mbit/s, instead of 3 Mbit/s as originally proposed in Section 6.2.

H1 is transmitting real-time traffic (video) with the absolute minimum requirement of 2 Mbit/s, but transmitting at a higher rate will ensure a higher video quality. We still want to define the flow's bandwidth requirement as 2 Mbit/s to free link resources which other flows can use, but we will try to avoid using 2 Mbit/s links as far as possible so the flow can use the additional free capacity when available. To do so, we add traffic class 1 to the policy, and Figure 6.7 displays the results.

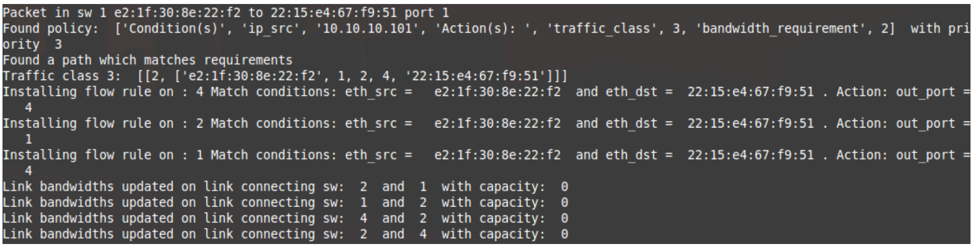
```
policy001 = policy_manager.Policy()
policy001.match(ip_src="10.10.10.101")
policy001.action(bandwidth_requirement=2,
allow_load_balance=True, traffic_class=1)
policy001.priority(5)
```



```
Packet in sw 1 e2:1f:30:8e:22:f2 to 22:15:e4:67:f9:51 port 1
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'traffic_class', 1, 'bandwidth_requirement', 2]  with pri
rity  3
Found a path which matches requirements
Traffic class 1:  [[8, ['e2:1f:30:8e:22:f2', 1, 4, '22:15:e4:67:f9:51']]]
Installing flow rule on : 4 Match conditions: eth_src =   e2:1f:30:8e:22:f2  and eth_dst =  22:15:e4:67:f9:51 . Action: out_port =
  4
Installing flow rule on : 1 Match conditions: eth_src =   e2:1f:30:8e:22:f2  and eth_dst =  22:15:e4:67:f9:51 . Action: out_port =
  5
Link bandwidths updated on link connecting sw:  4  and  1  with capacity:  2
Link bandwidths updated on link connecting sw:  1  and  4  with capacity:  2
```

Figure 6.7: Best capacity path taken

Explanation of Figure 6.7 and 6.8: We observe that by using traffic class 1, the application will install the flow on path s1-s4, whereby the link still has remaining capacity left after the flow added to the link. As long as no other flows reserve this capacity, it is free to be used by the flow. By changing the traffic class to 3 the opposite happens, and the application installs the flow using the path with the lowest bandwidth capacity.



```
Packet in sw 1 e2:1f:30:8e:22:f2 to 22:15:e4:67:f9:51 port 1
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'traffic_class', 3, 'bandwidth_requirement', 2]  with pri
ority  3
Found a path which matches requirements
Traffic class 3:  [[2, ['e2:1f:30:8e:22:f2', 1, 2, 4, '22:15:e4:67:f9:51']]]
Installing flow rule on : 4 Match conditions: eth_src =   e2:1f:30:8e:22:f2  and eth_dst =  22:15:e4:67:f9:51 . Action: out_port =
   4
Installing flow rule on : 2 Match conditions: eth_src =   e2:1f:30:8e:22:f2  and eth_dst =  22:15:e4:67:f9:51 . Action: out_port =
   1
Installing flow rule on : 1 Match conditions: eth_src =   e2:1f:30:8e:22:f2  and eth_dst =  22:15:e4:67:f9:51 . Action: out_port =
   4
Link bandwidths updated on link connecting sw:  2  and  1  with capacity:  0
Link bandwidths updated on link connecting sw:  1  and  2  with capacity:  0
Link bandwidths updated on link connecting sw:  4  and  2  with capacity:  0
Link bandwidths updated on link connecting sw:  2  and  4  with capacity:  0
```

Figure 6.8: Lowest capacity path taken

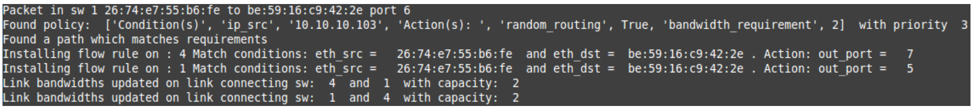
6.3.1 Summary

The SDN application is supposed to work autonomously, by taking the best path based on its calculations. Despite this fact, traffic classes can be used by the administrators to influence the application to take specific paths to some extent. The scenario shows that the controller generates different paths for the same flow (using same requirements) only by changing the traffic class.

6.4 Network Dynamics

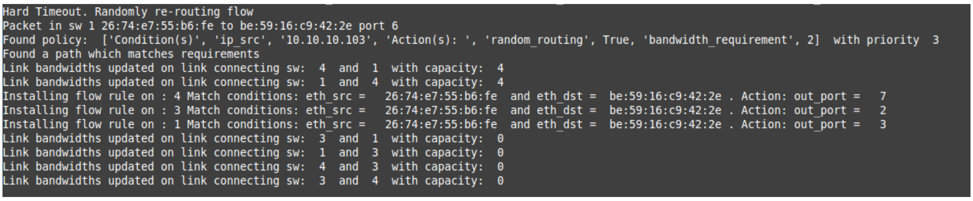
We assume that the previous policies have timed out. We want to initiate a new connection between h3 and h7. The flow does not need much bandwidth, but the transmitted data is of high interest for potential enemies. Consequently, we add randomness to the policy to prohibit predictable paths, as shown below:

```
policy003 = policy_manager.Policy()
policy003.match(ip_src="10.10.10.103")
policy003.action(bandwidth_requirement=2, random_routing=True)
policy003.priority(3)
```



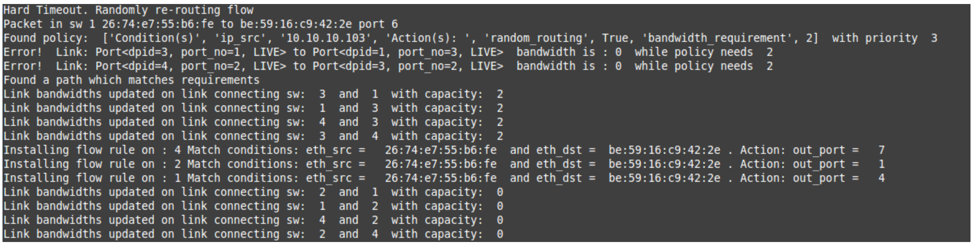
```
Packet in sw 1 26:74:e7:55:b6:fe to be:59:16:c9:42:2e port 6
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.103', 'Action(s): ', 'random_routing', True, 'bandwidth_requirement', 2]  with priority  3
Found a path which matches requirements
Installing flow rule on : 4 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   7
Installing flow rule on : 1 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   5
Link bandwidths updated on link connecting sw:  4  and  1  with capacity:  2
Link bandwidths updated on link connecting sw:  1  and  4  with capacity:  2
```

Figure 6.9: Chosen path for new incoming flow



```
Hard Timeout. Randomly re-routing flow
Packet in sw 1 26:74:e7:55:b6:fe to be:59:16:c9:42:2e port 6
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.103', 'Action(s): ', 'random_routing', True, 'bandwidth_requirement', 2]  with priority  3
Found a path which matches requirements
Link bandwidths updated on link connecting sw:  4  and  1  with capacity:  4
Link bandwidths updated on link connecting sw:  1  and  4  with capacity:  4
Installing flow rule on : 4 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   7
Installing flow rule on : 3 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   2
Installing flow rule on : 1 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   3
Link bandwidths updated on link connecting sw:  3  and  1  with capacity:  0
Link bandwidths updated on link connecting sw:  1  and  3  with capacity:  0
Link bandwidths updated on link connecting sw:  4  and  3  with capacity:  0
Link bandwidths updated on link connecting sw:  3  and  4  with capacity:  0
```

Figure 6.10: Chosen path for re-forwarded flow

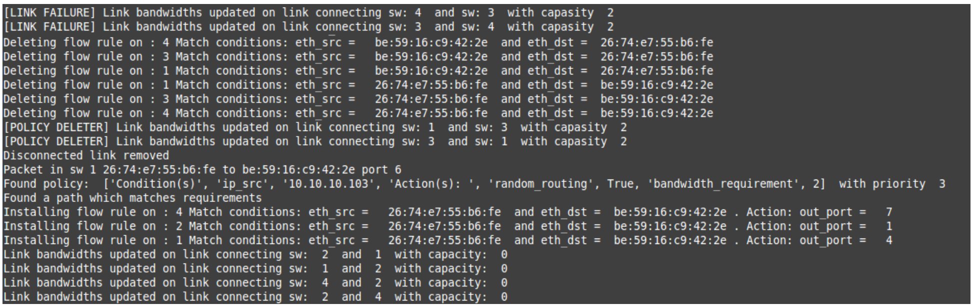


```
Hard Timeout. Randomly re-routing flow
Packet in sw 1 26:74:e7:55:b6:fe to be:59:16:c9:42:2e port 6
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.103', 'Action(s): ', 'random_routing', True, 'bandwidth_requirement', 2]  with priority  3
Error!  Link: Port<dpid=3, port_no=1, LIVE> to Port<dpid=1, port_no=3, LIVE>  bandwidth is : 0  while policy needs  2
Error!  Link: Port<dpid=4, port_no=2, LIVE> to Port<dpid=3, port_no=2, LIVE>  bandwidth is : 0  while policy needs  2
Found a path which matches requirements
Link bandwidths updated on link connecting sw:  3  and  1  with capacity:  2
Link bandwidths updated on link connecting sw:  1  and  3  with capacity:  2
Link bandwidths updated on link connecting sw:  4  and  3  with capacity:  2
Link bandwidths updated on link connecting sw:  3  and  4  with capacity:  2
Installing flow rule on : 4 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   7
Installing flow rule on : 2 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   1
Installing flow rule on : 1 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   4
Link bandwidths updated on link connecting sw:  2  and  1  with capacity:  0
Link bandwidths updated on link connecting sw:  1  and  2  with capacity:  0
Link bandwidths updated on link connecting sw:  4  and  2  with capacity:  0
Link bandwidths updated on link connecting sw:  2  and  4  with capacity:  0
```

Figure 6.11: Flow re-forwarded once more

Explanation of Figure 6.9, 6.10, and 6.11: The figures shows the paths used over a time period. When the flow rules exceed the hard-timeout limit, the controller knows that the flow is still active, but needs to be randomly re-forwarded.

The SDN application is also aware of link failures and ensures that the flows are re-forwarded when disconnections occur, as Figure 6.12 illustrates; where the link between s3 and s4 is disconnected while the flow is using the path.



```
[LINK FAILURE] Link bandwidths updated on link connecting sw: 4  and sw: 3  with capasity  2
[LINK FAILURE] Link bandwidths updated on link connecting sw: 3  and sw: 4  with capasity  2
Deleting flow rule on : 4 Match conditions: eth_src =   be:59:16:c9:42:2e  and eth_dst =  26:74:e7:55:b6:fe
Deleting flow rule on : 3 Match conditions: eth_src =   be:59:16:c9:42:2e  and eth_dst =  26:74:e7:55:b6:fe
Deleting flow rule on : 1 Match conditions: eth_src =   be:59:16:c9:42:2e  and eth_dst =  26:74:e7:55:b6:fe
Deleting flow rule on : 1 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e
Deleting flow rule on : 3 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e
Deleting flow rule on : 4 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e
[POLICY DELETER] Link bandwidths updated on link connecting sw: 1  and sw: 3  with capasity  2
[POLICY DELETER] Link bandwidths updated on link connecting sw: 3  and sw: 1  with capasity  2
Disconnected link removed
Packet in sw 1 26:74:e7:55:b6:fe to be:59:16:c9:42:2e port 6
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.103', 'Action(s): ', 'random_routing', True, 'bandwidth_requirement', 2]  with priority  3
Found a path which matches requirements
Installing flow rule on : 4 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   7
Installing flow rule on : 2 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   1
Installing flow rule on : 1 Match conditions: eth_src =   26:74:e7:55:b6:fe  and eth_dst =  be:59:16:c9:42:2e . Action: out_port =   4
Link bandwidths updated on link connecting sw:  2  and  1  with capacity:  0
Link bandwidths updated on link connecting sw:  1  and  2  with capacity:  0
Link bandwidths updated on link connecting sw:  4  and  2  with capacity:  0
Link bandwidths updated on link connecting sw:  2  and  4  with capacity:  0
```

Figure 6.12: Link disconnecting while flow using it

Explanation of Figure 6.12: The controller detects the disconnected link and rapidly re-forwards the flow using another path. The disconnected link is excluded from the random forwarding pool until it becomes active again.

6.4.1 Summary

Based on a predefined period, the application will perform re-forwarding of flows when reaching the time threshold. The application uses a random path algorithm to generate a new path which achieves the bandwidth requirements, but where the application excludes the last used path from the calculation.

When a link in use is disconnected, then the application is notified, whereby it will iterate over the policies which use paths which includes the broken link. Following this, the controller will then remove all the flow rules for these corresponding flows, of which will result in a re-processing by the controller where it will generate alternative paths for the flows.

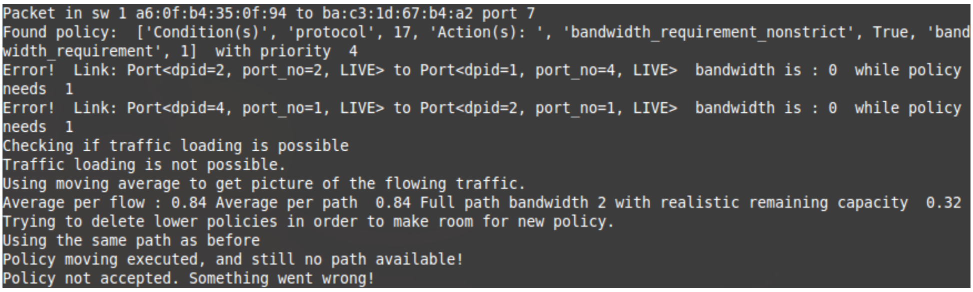
6.5 Monitoring

This scenario is a continuation of the scenario presented in the previous section, and policy003 is still active on path s1-s2-s4. In this situation, the link between s1 and s4 has also been disconnected, leaving only one path (via s1-s2-s4) open across the network.

We create a new policy saying UDP traffic should get a bandwidth requirement of 1 Mbit/s, independent of which host is transmitting. We do this because we know real-time traffic in forms of UDP has higher priority than TCP for this network. However, the requirement is set to non-strict, so that the flows can handle a temporary lower bandwidth than the specified requirement. We start a UDP session between h4 and h8 with the following policy added to the policy list:

```
policy004 = policy_manager.Policy()
policy004.match(protocol=17)
policy004.action(bandwidth_requirement=1,
                 bandwidth_requirement_nonstrict=True)
policy004.priority(4)
```

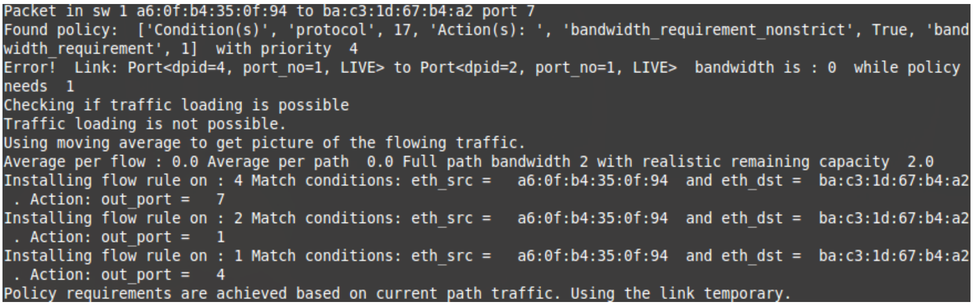
Explanation of Figure 6.13: In this scenario there is a challenge. h3 (using policy003) has reserved 2 Mbit/s on the path, leaving 0 Mbit/s left for h4, which is below the UDP requirement. Policy003 also has the highest priority. The UDP policy is defined as non-strict, meaning the flow can accept lower bandwidths than 1 Mbit/s for an arbitrarily period. The SDN application takes decisions depending on how much of the physical link in use by monitoring the ongoing traffic. In this scenario, the application denies the flow access to the network, because the monitored traffic is too high.



```
Packet in sw 1 a6:0f:b4:35:0f:94 to ba:c3:1d:67:b4:a2 port 7
Found policy:  ['Condition(s)', 'protocol', 17, 'Action(s): ', 'bandwidth_requirement_nonstrict', True, 'band
width_requirement', 1]  with priority  4
Error!  Link: Port<dpid=2, port_no=2, LIVE> to Port<dpid=1, port_no=4, LIVE>  bandwidth is : 0  while policy
needs  1
Error!  Link: Port<dpid=4, port_no=1, LIVE> to Port<dpid=2, port_no=1, LIVE>  bandwidth is : 0  while policy
needs  1
Checking if traffic loading is possible
Traffic loading is not possible.
Using moving average to get picture of the flowing traffic.
Average per flow : 0.84 Average per path  0.84 Full path bandwidth 2 with realistic remaining capacity  0.32
Trying to delete lower policies in order to make room for new policy.
Using the same path as before
Policy moving executed, and still no path available!
Policy not accepted. Something went wrong!
```

Figure 6.13: Denied flow

On the other hand, if the link is idle the following happens:



```
Packet in sw 1 a6:0f:b4:35:0f:94 to ba:c3:1d:67:b4:a2 port 7
Found policy:  ['Condition(s)', 'protocol', 17, 'Action(s): ', 'bandwidth_requirement_nonstrict', True, 'band
width_requirement', 1]  with priority  4
Error!  Link: Port<dpid=4, port_no=1, LIVE> to Port<dpid=2, port_no=1, LIVE>  bandwidth is : 0  while policy
needs  1
Checking if traffic loading is possible
Traffic loading is not possible.
Using moving average to get picture of the flowing traffic.
Average per flow : 0.0 Average per path  0.0 Full path bandwidth 2 with realistic remaining capacity  2.0
Installing flow rule on : 4 Match conditions: eth_src =   a6:0f:b4:35:0f:94  and eth_dst =  ba:c3:1d:67:b4:a2
 . Action: out_port =   7
Installing flow rule on : 2 Match conditions: eth_src =   a6:0f:b4:35:0f:94  and eth_dst =  ba:c3:1d:67:b4:a2
 . Action: out_port =   1
Installing flow rule on : 1 Match conditions: eth_src =   a6:0f:b4:35:0f:94  and eth_dst =  ba:c3:1d:67:b4:a2
 . Action: out_port =   4
Policy requirements are achieved based on current path traffic. Using the link temporary.
```

Figure 6.14: Approved flow

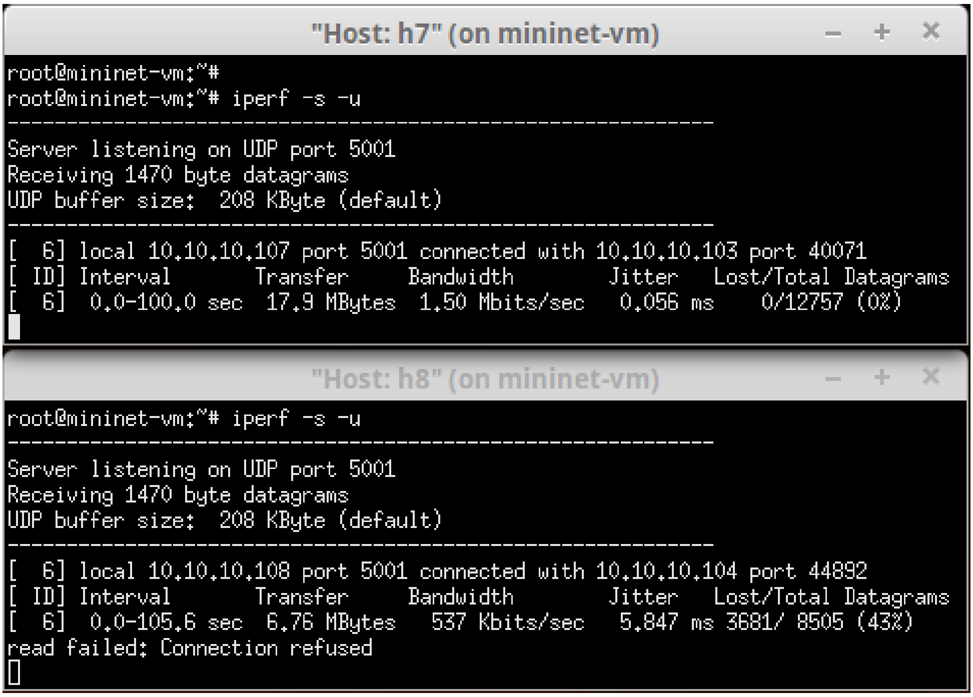
Explanation of Figure 6.14: The controller monitors the average traffic from h3, and do approvals if the average traffic is below certain security thresholds. In this case, the flow is added to the path because the traversing traffic is little.

Most importantly, when adding the flow to the link, what will happen if h3 starts transmitting again, in a manner that exceeds the links capacity? Figure 6.15 displays the process[2] , where h3 starts transmitting at a rate of 1.5 Mbit/s and h4 in 1 Mbit/s, which is 0.5 Mbit/s more than the link capacity.

Explanation of Figure 6.15: The figure displays that the traffic is policed to the links maximum bandwidth limit, but only h4-h8 experience loss, while h3-h7 is not affected.

When installing flow rules, the controller pushes strict policies and non-strict policies in different QoS queues at the switches. When reaching the links maximum capacity limits, the switches will start policing of the flows. Strict policies use a queue with

[2] Commands used to generate traffic: H3: *perf -c 10.10.10.107 -u -b 1.5M -t 100.* H4: *iperf -c 10.10.10.108 -u -b 1M -t 100.*



```
root@mininet-vm:~#
root@mininet-vm:~# iperf -s -u
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  6] local 10.10.10.107 port 5001 connected with 10.10.10.103 port 40071
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  6]  0.0-100.0 sec  17.9 MBytes  1.50 Mbits/sec   0.056 ms    0/12757 (0%)
```

```
root@mininet-vm:~# iperf -s -u
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  6] local 10.10.10.108 port 5001 connected with 10.10.10.104 port 44892
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  6]  0.0-105.6 sec  6.76 MBytes   537 Kbits/sec   5.847 ms 3681/ 8505 (43%)
read failed: Connection refused
```

Figure 6.15: Exceeding the link capacity

a higher requirement than the non-strict ones, therefore, will the non-strict queue always have to give way for the strict queue, which leads to packet drops while the strict queue is unaffected.

6.5.1 Summary

The first example shows a flow which is denied access to the network, because the path is fully reserved, and the actual traffic (obtained by monitoring the flowing traffic) on the path is using significant parts of the bandwidth. The application is using an algorithm[3] to estimate how much capacity which will be in use if the new flow is added, by computing average bandwidth on the path and average bandwidth per flow. In the first example, the controller is considering the path to be full, while the second example is approved, because even tough the link is fully reserved the actual traffic is close to none. The result is that the flow is allowed to use the link.

On the other hand, it is not scalable to add flows to to fully reserved paths, just because the flows are not utilizing their reserved bandwidth on an average basis. The point of using bandwidth requirements is to achieve QoS for the particular flows, but again, we also want to utilize the network as much as possible. The third example is testing of such a scenario: we have added flows with a higher aggregated bandwidth requirement than the link capacity. We start transmitting higher load than the link capacity which results in that the temporarily added flows are policed, while the first policies reserved on the link still achieve their bandwidth requirement (ensuring QoS).

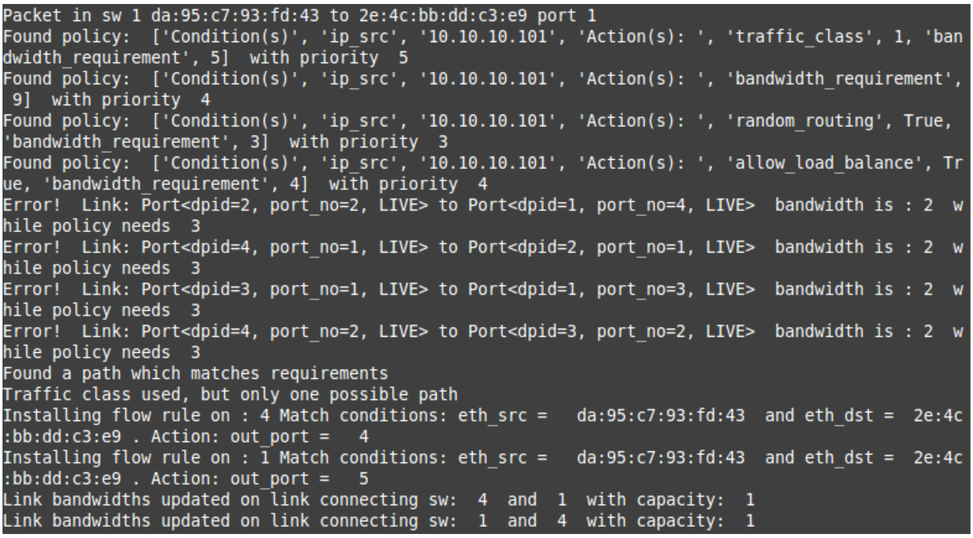
6.6 Policy Management

To validate the management properties of the application, we want to test some of the facing challenges. Many administrators would in practice most likely manage the policy application, and there could be instances where policies are conflicting with each other. We want to test how the controller chooses a policy when defining multiple policies with corresponding match conditions. We add new policies with the IP address of h1 as match condition with different priorities, and issue a ping to another host in the network.

```
policy005.match(ip_src="10.10.10.101")
policy005.action(bandwidth_requirement=5, traffic_class=1)
policy005.priority(5)

policy006.match(ip_src="10.10.10.101")
policy006.action(bandwidth_requirement=9)
policy006.priority(4)

policy007.match(ip_src="10.10.10.101")
```

[3] The algorithms is presented in Section 7.5

```
policy007.action(bandwidth_requirement=3, random_routing=True)
policy007.priority(3)

policy008.match(ip_src="10.10.10.101")
policy008.action(bandwidth_requirement=4, allow_load_balance=True)
policy008.priority(4)
```



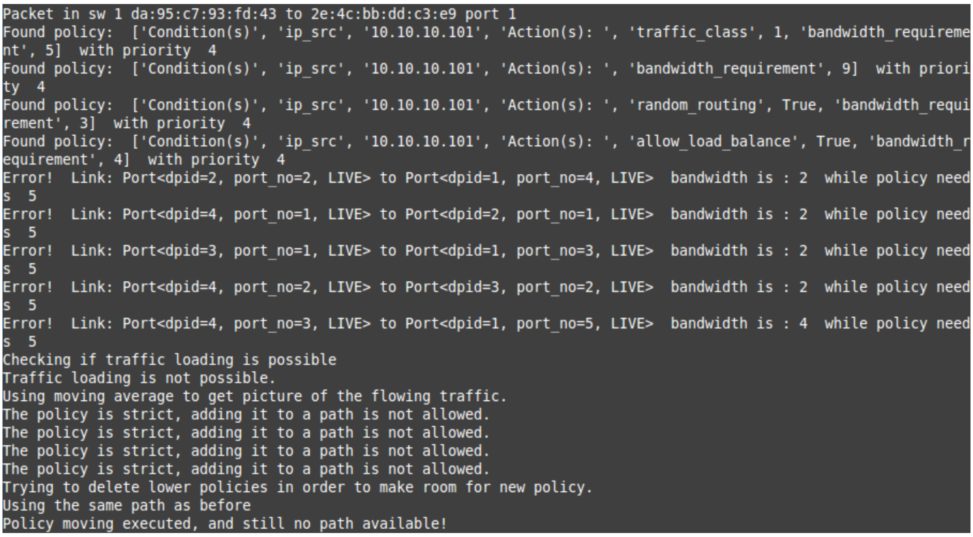
```
Packet in sw 1 da:95:c7:93:fd:43 to 2e:4c:bb:dd:c3:e9 port 1
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'traffic_class', 1, 'ban
dwidth_requirement', 5]  with priority  5
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'bandwidth_requirement',
 9]  with priority  4
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'random_routing', True,
'bandwidth_requirement', 3]  with priority  3
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'allow_load_balance', Tr
ue, 'bandwidth_requirement', 4]  with priority  4
Error!  Link: Port<dpid=2, port_no=2, LIVE> to Port<dpid=1, port_no=4, LIVE>  bandwidth is : 2  w
hile policy needs  3
Error!  Link: Port<dpid=4, port_no=1, LIVE> to Port<dpid=2, port_no=1, LIVE>  bandwidth is : 2  w
hile policy needs  3
Error!  Link: Port<dpid=3, port_no=1, LIVE> to Port<dpid=1, port_no=3, LIVE>  bandwidth is : 2  w
hile policy needs  3
Error!  Link: Port<dpid=4, port_no=2, LIVE> to Port<dpid=3, port_no=2, LIVE>  bandwidth is : 2  w
hile policy needs  3
Found a path which matches requirements
Traffic class used, but only one possible path
Installing flow rule on : 4 Match conditions: eth_src =   da:95:c7:93:fd:43  and eth_dst =  2e:4c
:bb:dd:c3:e9 . Action: out_port =   4
Installing flow rule on : 1 Match conditions: eth_src =   da:95:c7:93:fd:43  and eth_dst =  2e:4c
:bb:dd:c3:e9 . Action: out_port =   5
Link bandwidths updated on link connecting sw:  4  and  1  with capacity:  1
Link bandwidths updated on link connecting sw:  1  and  4  with capacity:  1
```

Figure 6.16: The policy with highest priority is chosen (policy007)

Explanation of Figure 6.16: The application will match the flow with all of the policies. The application will then sort the policies according to their priorities, and then execute the highest prioritized policy while throwing the other ones.

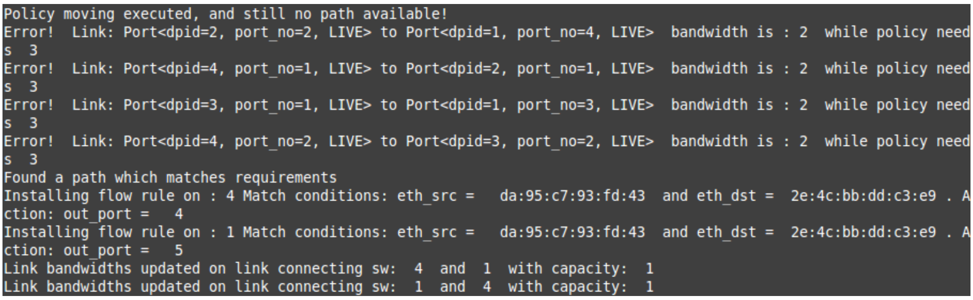
However, what happens if the policies have the same priority? We change all the policies to use 4 as priority and issue a ping request.

Explanation of Figure 6.17 and 6.18: The application is unable to sort the policies correctly since they all share the same priority, thus starts arbitrary with one policy and test it against the network. The application will iterate through all policies until it finds an approved one, then install it. Figure 6.17 shows that the application tries to install policy005, but the policy is denied because of its requirements. The next policy in line is policy007, and it is approved and installed.



```
Packet in sw 1 da:95:c7:93:fd:43 to 2e:4c:bb:dd:c3:e9 port 1
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'traffic_class', 1, 'bandwidth_requireme
nt', 5]  with priority  4
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'bandwidth_requirement', 9]  with priori
ty  4
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'random_routing', True, 'bandwidth_requi
rement', 3]  with priority  4
Found policy:  ['Condition(s)', 'ip_src', '10.10.10.101', 'Action(s): ', 'allow_load_balance', True, 'bandwidth_r
equirement', 4]  with priority  4
Error!  Link: Port<dpid=2, port_no=2, LIVE> to Port<dpid=1, port_no=4, LIVE>  bandwidth is : 2  while policy need
s  5
Error!  Link: Port<dpid=4, port_no=1, LIVE> to Port<dpid=2, port_no=1, LIVE>  bandwidth is : 2  while policy need
s  5
Error!  Link: Port<dpid=3, port_no=1, LIVE> to Port<dpid=1, port_no=3, LIVE>  bandwidth is : 2  while policy need
s  5
Error!  Link: Port<dpid=4, port_no=2, LIVE> to Port<dpid=3, port_no=2, LIVE>  bandwidth is : 2  while policy need
s  5
Error!  Link: Port<dpid=4, port_no=3, LIVE> to Port<dpid=1, port_no=5, LIVE>  bandwidth is : 4  while policy need
s  5
Checking if traffic loading is possible
Traffic loading is not possible.
Using moving average to get picture of the flowing traffic.
The policy is strict, adding it to a path is not allowed.
The policy is strict, adding it to a path is not allowed.
The policy is strict, adding it to a path is not allowed.
The policy is strict, adding it to a path is not allowed.
Trying to delete lower policies in order to make room for new policy.
Using the same path as before
Policy moving executed, and still no path available!
```

Figure 6.17: Iterating through the policies



```
Policy moving executed, and still no path available!
Error!  Link: Port<dpid=2, port_no=2, LIVE> to Port<dpid=1, port_no=4, LIVE>  bandwidth is : 2  while policy need
s  3
Error!  Link: Port<dpid=4, port_no=1, LIVE> to Port<dpid=2, port_no=1, LIVE>  bandwidth is : 2  while policy need
s  3
Error!  Link: Port<dpid=3, port_no=1, LIVE> to Port<dpid=1, port_no=3, LIVE>  bandwidth is : 2  while policy need
s  3
Error!  Link: Port<dpid=4, port_no=2, LIVE> to Port<dpid=3, port_no=2, LIVE>  bandwidth is : 2  while policy need
s  3
Found a path which matches requirements
Installing flow rule on : 4 Match conditions: eth_src =   da:95:c7:93:fd:43  and eth_dst =  2e:4c:bb:dd:c3:e9 . A
ction: out_port =   4
Installing flow rule on : 1 Match conditions: eth_src =   da:95:c7:93:fd:43  and eth_dst =  2e:4c:bb:dd:c3:e9 . A
ction: out_port =   5
Link bandwidths updated on link connecting sw:  4  and  1  with capacity:  1
Link bandwidths updated on link connecting sw:  1  and  4  with capacity:  1
```

Figure 6.18: Approved policy

6.6.1 Summary

In this scenario, we have many policies which apply to the same flow, but the controller executes the policy with the highest priority. In practice, the controller picks up every policy and stores them in a list sorted by priority, where enforcing the highest ones first. For the next scenario, the controller will iterate through the policies until it finds a policy of which is approved by the network.

Chapter 7 Experiences from the Implementation

This chapter is based on knowledge gained from the laboratory implementation. Represented are different scenarios, challenges, and practical solutions.

7.1 Enforcing Policies

We chose to compile policies into flow rules where the match conditions in the policy may differ from the match conditions in the flow rules. The match conditions in the policies can vary while the flow rules are always installed on the switches using the combination of MAC source and destination address. The reason for this approach is to make the rules unique per direction of a flow, to force all flows to be processed by the controller because the application installs rules by the reactive approach. We underline our design decision by an example.

Let us say we have many specific policies in our policy list, but we want to add a wide one (with the lowest priority) saying that UDP traffic should use a distinct path in the network. The main thought behind this is that every UDP flow of which does not match with the other specified policies then should be matched with this policy, due to it has a very low priority. By converting this policy into a wide flow rule where the match condition is set to the UDP, then all incoming UDP flows will have a match with this rule independent of other parameters.

The flow rule becomes a problem when the controller installs the rule before the other rules due to it being so wide. The UDP flows will always end up being forwarded by the switches, even though the application possesses higher priority policies for the incoming flows of which is waiting to be enforced. The switches will never forward the flows to the controller for processing because of the match with the flow rule. The remaining policies, will therefore never be applied as long as the UDP flow rule is active.

On the other hand, we acknowledge weaknesses with our approach, by only using the MAC source and destination as match conditions. By using this method, the application only allows one policy to work for a flow direction for a particular communication pair. Let us use the same example where UDP traffic should use a particular path in the network. A particular flow enters the network and is processed by the controller and matches with the UDP condition. The application then compiles the policy to flow rules where the match condition is the combination of MAC source and destination in the flow rule. This will allow all other flows to be processed by the controller because of no one will match with that particular flow rule. More importantly, they will be enforced by the same policy by the controller, but at the SDN forwarding plane; the communication pairs are matched on a unique basis. However, if the original communication pair suddenly switches over to TCP traffic, they will still have a match with the installed flow rules. This is due to that the protocol does not have any significance for the match conditions in the flow rules because they will contain using the same MAC addresses. The developed application uses flow rule timeouts (flushing) as a semi-mitigation to this problem, to ensure that flow rules are fresh and to let the application regularly process the flows.

Instead, we could have mitigated this better by including a third match criterion (example protocol) in the match condition additional to the source and destination MAC addresses. If this were the case, then the flow would have been forwarded to the controller for processing when changing to TCP traffic, because of no matching flow rules. The key element when using a reactive approach is to narrow the rules down, so that the flows does not match with extensive match conditions, in a way that prohibits the flows from being processed by the controller.

7.2 ARP Broadcasts

In legacy switching, ARP broadcasts are used to discover entities in the network, but this can lead to problems in topologies containing loops. To avoid broadcast loops, a protocol called Spanning Tree Protocol (STP) [7] is used to build a topology tree with branches to every switch without loops. Ryu has a module for STP which can be included in custom made code, but we wanted the controller to be independent of this module. By using the controllers capabilities to obtain topology information about links, we were able to create a type of topology abstraction to forward ARP broadcasts without causing loops in the network.

Figure 7.1 illustrates the ARP process. The function abstracts the links connected between switches, and only let the remaining links be visible for the application. Instead of broadcasting the packets from switch to switch, the controller iterates over all the switches that still has a visible link (if the switch still has a link after the logical link removal, then we assume that it has a host connected to it). The application then instructs the switch to create a new ARP request containing the

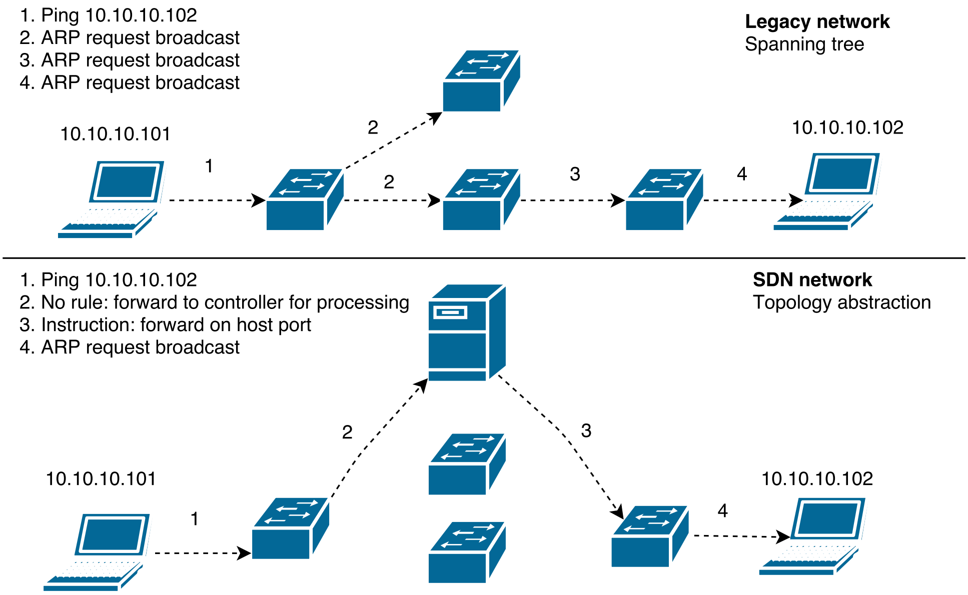


Figure 7.1: Legacy vs SDN ARP forwarding

same properties as the request that the controller got in, and then forwards the request out the remaining links. The advantage of this solution is it generates low traffic on the network, by removing broadcast from switch to switch. The abstraction is a many-to-one approach, where all the switches in the network are viewed as one big switch because the intermediate switch-links are abstracted, and only the links connected to hosts are used to transmit the ARP request.

7.3 Consistency

One of the main challenges in the laboratory implementation was to keep state about ongoing flows and policies. To enforce resource reservations, the controller is dependable of saving and updating its resource pool when installing the flows. The reservations are not stored in the switches but only by the controller. The reservations must, thus, be updated when flow rules times out on the switches to maintain consistency between the applications view of the reservations (state of resource pool) and the actual reservations (flow rules). We solved this by introducing a flag in every flow rule:

```
flags=ofproto_v1_3.OFPFF_SEND_FLOW_REM
```

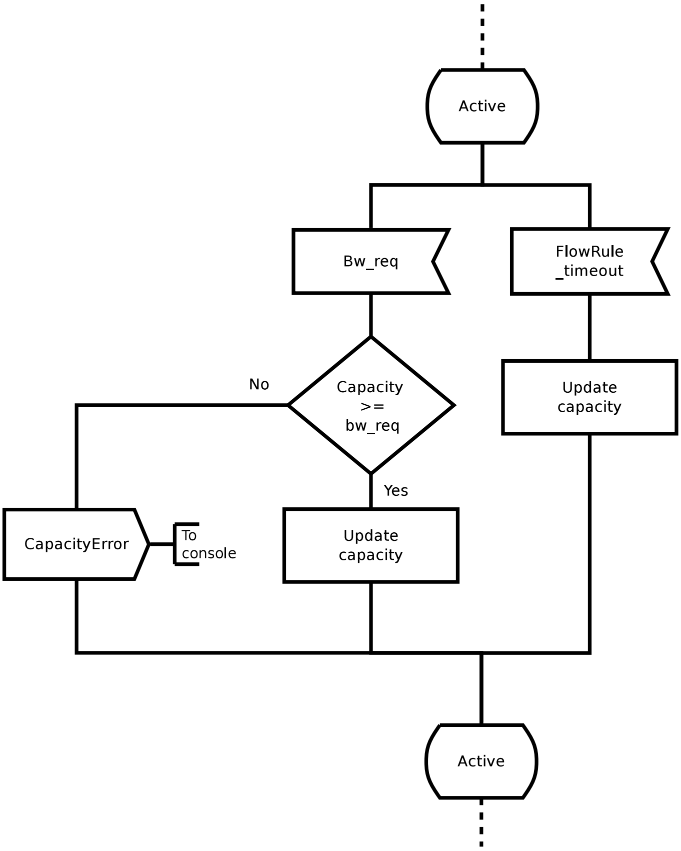


Figure 7.2: Example of the capacity update process

This flag gives notice to the controller when a flow rule times out. The controller will use this to update the resource pool. Figure 7.2 illustrates a simplified SDL process diagram of the procedure: The controller listens for incoming flows with bandwidth requirements or flow rule timeouts, and updates its resource pool accordingly. A flow rule timeout indicates that the policy is no longer running on the network. Despite this, the policy is also removed from the *running policies*-list (ref 5.3.1) when timing out.

Using this approach introduces potential weaknesses. Link failures or congestion could cause problems in scenarios where the controller does not get the updates from the switches: the controller may think the link is fully utilized even though the flow rules, in reality, has timed out. A mitigation strategy could be to perform regular flushing of the whole system and build it up again in a reactive manner.

7.4 Flushing

We also use the timeout-flag to ensure that the network is regularly flushed and utilized to the maximum. The flag is triggered both at idle-timeout and at hard-timeout. Where the idle-timeout is triggered when the flow is unused for a given period and ensures that the flow rule is deleted without the need of help from the controller, and is a network utilization strategy by freeing unused reserved resources.

Hard-timeouts, on the other hand, are used to ensure freshness to the flows. If new links are discovered and added to the network topology, the desire is to use these additional links to utilize the network. The application does not perform instant re-forwarding when this occur, but because of the implemented hard-timeouts, the flows are forced through a re-processing by the application when the hard-timeout is triggered. The re-processing uses the newly updated topology as input to path calculation. The SDN application uses hard-timeouts as a basis for performing the regular random routing of flows if this is specified in the policy.

7.5 QoS

We wanted to utilize the network as much as possible, but to fulfill QoS requirements, the controller can not add more policies to the link if the aggregated policy bandwidth requirements are equal to the links maximum capacity. Even though the traversing traffic may be a lot lower than the reserved requirements; we must consider busy hours[1] where the link is fully utilized.

In contrast, we wanted to utilize the link when there is no busy hour. To do so, we introduce monitoring, QoS queues, and non-strict policies. The non-strict property must be applied to the policy to state that the flow tolerates a lower requirement that what is specified in the policy, thus stating that it does not need QoS. Monitoring is used to find out how much traffic is traversing down the different paths[2], and adds the input to an algorithm that calculates the moving average for each path and each flow. When a non-strict policy added, it will need to pass the following test to use the link temporary:

Code sequence 7.1 Capacity test of non-strict policies

```
capacity = full_path - (average_path + average_flow)
if capacity >= full_path/2 and capacity >= req:
```

[1] Busy hour refers to the period where the network traffic load is at maximum.

[2] Monitoring is performed by regularly inspecting sent and received packets on ports.

The test takes the average traffic flowing on the path added together with the average traffic per flow (represents the expected traffic by adding a new flow), and checks if less than 50% of the realistic[3] path capacity is in use by adding the new flow, and as well that the realistic capacity is higher that the actual policy requirement. The checks are performed to ensure that the link has a buffer so that the risk of link congestion after adding in the new flow is low.

Additional to this, the temporary flow is placed in a QoS queue with lower requirement than the strict policies. We define queues by the following instructions:

Code sequence 7.2 Queue generation

```
sudo ovs-vsctl set port s1-eth1 qos=@defaultqos -- --id=@defaultqos
create qos type=linux-htb other-config:max-rate=link_bandwidth
queues=0=@q0,1=@q1 --
"--id=@q0 create queue other-config:min-rate=link_bandwidth -- " \
"--id=@q1 create queue other-config:min-rate=0"
```

q0 has *min-rate* set to the links maximum bandwidth capacity while q1 has no minimum limit. The result in Figure 6.15 showed that q1 were policed while q0 was unaffected when the aggregated transmitted traffic exceeded the links bandwidth capacity. By installing ingress port policing and placement of flows in different queues we can ensure QoS to particular flows. There are additional approaches of ensuring QoS with queues, such as to place flows in individual queues with different maximum rate; this will also ensure policing of traffic.

Figure 7.3 displays the overall process: The controller adds the extra flow which is exceeding the paths capacity, due to the monitored traffic is significantly lower than the reservations. By computations, the controller believes that the extra added flow will achieve its requirements on an average basis, but it can not guarantee QoS for that particular flow.

Flows without any matching policies will always be placed in q1, so they have to give away to the strict policies.

[3] The realistic capacity is the monitored available capacity

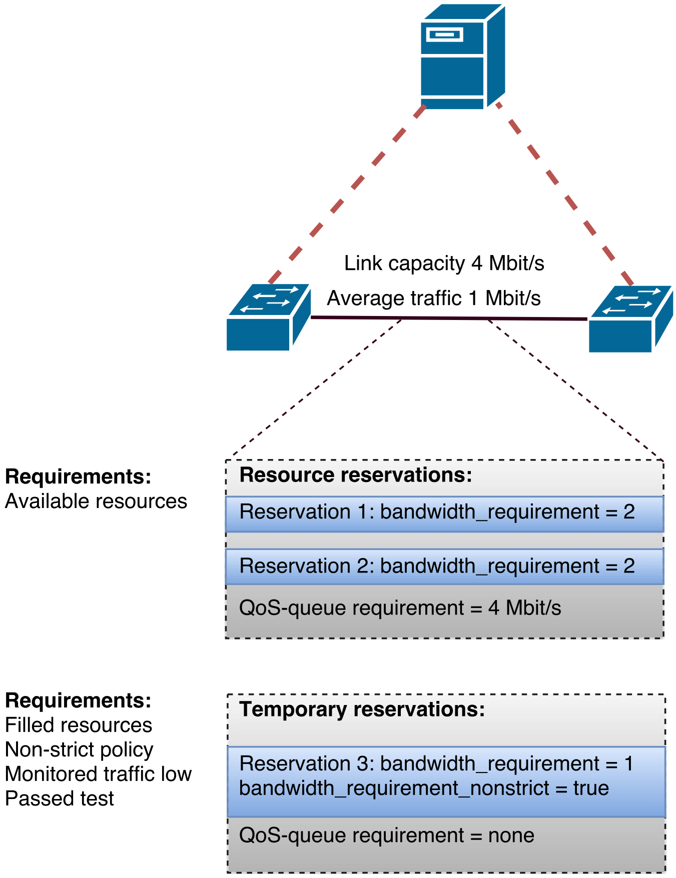

Link capacity 4 Mbit/s
Average traffic 1 Mbit/s
Requirements:
Available resources
Resource reservations:
Reservation 1: bandwidth_requirement = 2
Reservation 2: bandwidth_requirement = 2
QoS-queue requirement = 4 Mbit/s
Requirements:
Filled resources
Non-strict policy
Monitored traffic low
Passed test
Temporary reservations:
Reservation 3: bandwidth_requirement = 1
bandwidth_requirement_nonstrict = true
QoS-queue requirement = none

Figure 7.3: Adding policies beyond capability limitations

7.6 Traffic Loading

Group rules are used for splitting traffic in scenarios where the flow needs to be load balanced to achieve the policy bandwidth requirement. The controller iterates over the paths and calculates how much of the bandwidth the flow shall seize, for example, 1 Mbit/s from path X and 2 Mbit/s from path Y, while combined satisfying a bandwidth requirement of 3 Mbit/s. By introducing weights, we can adjust the traffic to flow according to the seized link capacities. The thought behind this feature was to perform load balancing to ongoing traffic by using multiple ports at the same time. However, we experienced that the traffic loading feature did not work as expected.

It turns out that group rules (using the *SELECT* option with weights [15]) do not have the ability to use multiple ports to forward selected parts of the traffic at once[4], they rather switch between using the individual ports based on a switch-computed selection algorithm. When testing the traffic loading function we experienced that when initiating the traffic generation with *iperf*, the group rule will use the weight to choose a port. In spite of this, once the port is selected, the switch will use the port for the entire transmission period. It emerges as the port used is based on a timer, and if the transmitting traffic is continuous, the same port will be used throughout the session, and never use both ports simultaneously for the session. On the other hand, Figure 7.4 displays that the switch can use multiple ports to send various session traffic. By using two remote connections to h1, and by generating ping and iperf traffic to h5, we see that s1 forwards the traffic via different ports by the group rule. The ping-traffic uses path s1-s2-s4 while iperf traffic uses s1-s3-s4. The timestamps in the figure show that traffic loading is happening simultaneously.

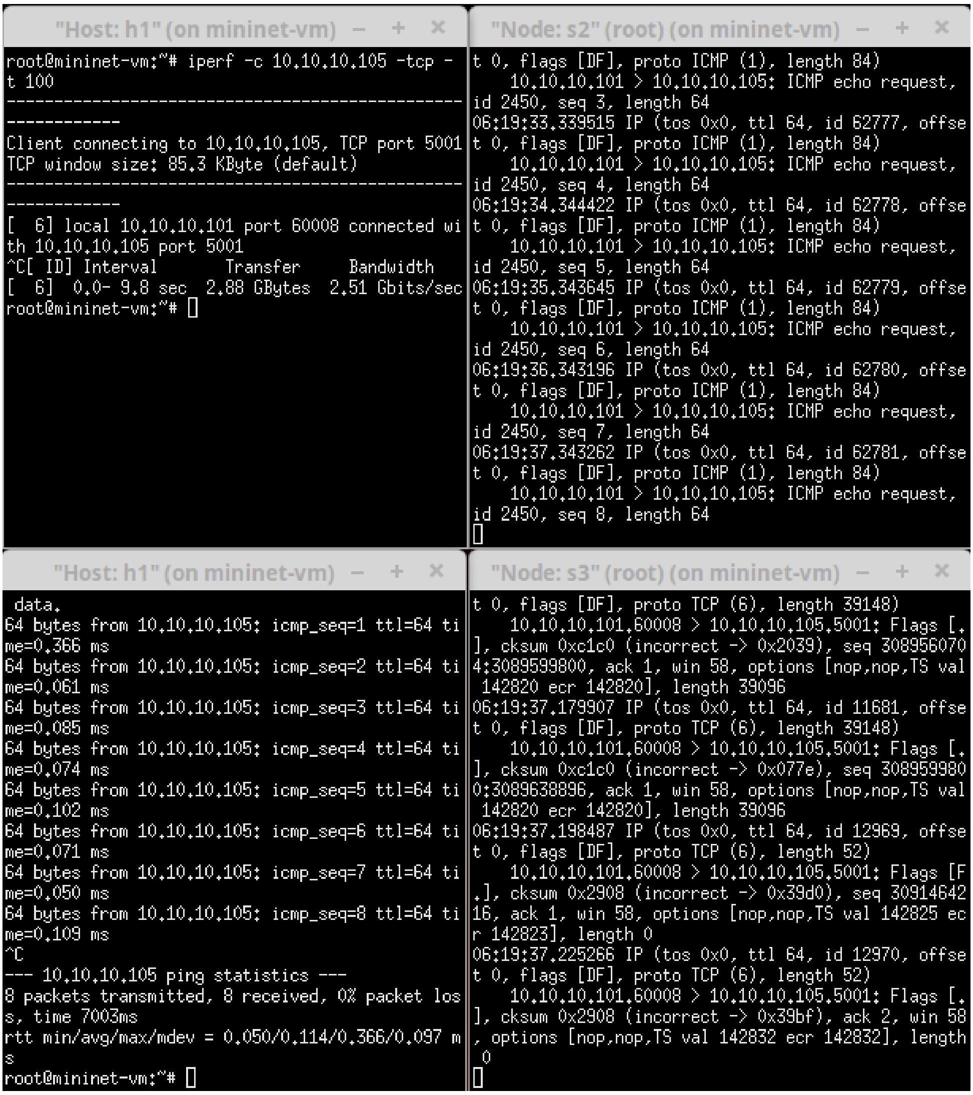
As a result of the group rule limitation, the bandwidth reservation in Section 6.2 is invalid. For this particular scenario, we used the traffic loading feature on s1 to split an incoming flow of 3 Mbit/s down into 1 Mbit/s on port 3 and 2 Mbit/s on port 4. The goal was to retain the same throughput in as out, by using multiple ports to achieve an aggregated bandwidth of 3 Mbit/s for a particular flow. Instead, if h1 starts transmitting a session at 3 Mbit/s, the traffic will only use one of the ports (based on weighting), and therefore be policed according to that particular links capacity. For the validated scenario, this would imply QoS for maximum 2 Mbit/s via port 4, whereby the remaining 1 Mbit/s is either unused or must be used by another session. We have yet to identify opportunities of splitting particular parts of the same session by using multiple active ports in SDN or technical documentation about the switch-computed selection algorithm used in the group rules.

On the other hand, apart from being used with bandwidth reservations, the feature is an efficient way, in the long run, to ensure utilizing of the links with SDN. Figure 7.5 shows the result on s1 port basis[5] after 10 runs with iperf[6]. The loaded traffic (tx_packets) will slowly converge to the relative weights in the group rule.

[4] For example, to load balance an incoming flow onto multiple outgoing ports to retain the same throughput.

[5] Command used on s1: *sudo ovs-ofctl -O OpenFlow13 dump-ports s1.*

[6] Command used on h1: *iperf -c 10.10.10.105 -tcp -t 5.*


"Host: h1" (on mininet-vm)
"Node: s2" (root) (on mininet-vm)
"Host: h1" (on mininet-vm)
"Node: s3" (root) (on mininet-vm)

```
root@mininet-vm:~# iperf -c 10.10.10.105 -tcp -
t 100
------------------------------------------------
------------
Client connecting to 10.10.10.105, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------
------------
[  6] local 10.10.10.101 port 60008 connected wi
th 10.10.10.105 port 5001
^C[ ID] Interval       Transfer     Bandwidth
[  6]  0.0- 9.8 sec  2.88 GBytes  2.51 Gbits/sec
root@mininet-vm:~# 
```

```
t 0, flags [DF], proto ICMP (1), length 84)
    10.10.10.101 > 10.10.10.105: ICMP echo request,
id 2450, seq 3, length 64
06:19:33.339515 IP (tos 0x0, ttl 64, id 62777, offse
t 0, flags [DF], proto ICMP (1), length 84)
    10.10.10.101 > 10.10.10.105: ICMP echo request,
id 2450, seq 4, length 64
06:19:34.344422 IP (tos 0x0, ttl 64, id 62778, offse
t 0, flags [DF], proto ICMP (1), length 84)
    10.10.10.101 > 10.10.10.105: ICMP echo request,
id 2450, seq 5, length 64
06:19:35.343645 IP (tos 0x0, ttl 64, id 62779, offse
t 0, flags [DF], proto ICMP (1), length 84)
    10.10.10.101 > 10.10.10.105: ICMP echo request,
id 2450, seq 6, length 64
06:19:36.343196 IP (tos 0x0, ttl 64, id 62780, offse
t 0, flags [DF], proto ICMP (1), length 84)
    10.10.10.101 > 10.10.10.105: ICMP echo request,
id 2450, seq 7, length 64
06:19:37.343262 IP (tos 0x0, ttl 64, id 62781, offse
t 0, flags [DF], proto ICMP (1), length 84)
    10.10.10.101 > 10.10.10.105: ICMP echo request,
id 2450, seq 8, length 64
```

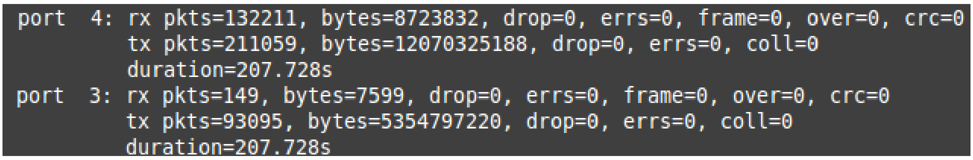
```
 data.
64 bytes from 10.10.10.105: icmp_seq=1 ttl=64 ti
me=0.366 ms
64 bytes from 10.10.10.105: icmp_seq=2 ttl=64 ti
me=0.061 ms
64 bytes from 10.10.10.105: icmp_seq=3 ttl=64 ti
me=0.085 ms
64 bytes from 10.10.10.105: icmp_seq=4 ttl=64 ti
me=0.074 ms
64 bytes from 10.10.10.105: icmp_seq=5 ttl=64 ti
me=0.102 ms
64 bytes from 10.10.10.105: icmp_seq=6 ttl=64 ti
me=0.071 ms
64 bytes from 10.10.10.105: icmp_seq=7 ttl=64 ti
me=0.050 ms
64 bytes from 10.10.10.105: icmp_seq=8 ttl=64 ti
me=0.109 ms
^C
--- 10.10.10.105 ping statistics ---
8 packets transmitted, 8 received, 0% packet los
s, time 7003ms
rtt min/avg/max/mdev = 0.050/0.114/0.366/0.097 m
s
root@mininet-vm:~# 
```

```
t 0, flags [DF], proto TCP (6), length 39148)
    10.10.10.101.60008 > 10.10.10.105.5001: Flags [.
], cksum 0xc1c0 (incorrect -> 0x2039), seq 308956070
4:3089599800, ack 1, win 58, options [nop,nop,TS val
 142820 ecr 142820], length 39096
06:19:37.179907 IP (tos 0x0, ttl 64, id 11681, offse
t 0, flags [DF], proto TCP (6), length 39148)
    10.10.10.101.60008 > 10.10.10.105.5001: Flags [.
], cksum 0xc1c0 (incorrect -> 0x077e), seq 308959980
0:3089638896, ack 1, win 58, options [nop,nop,TS val
 142820 ecr 142820], length 39096
06:19:37.198487 IP (tos 0x0, ttl 64, id 12969, offse
t 0, flags [DF], proto TCP (6), length 52)
    10.10.10.101.60008 > 10.10.10.105.5001: Flags [F
.], cksum 0x2908 (incorrect -> 0x39d0), seq 30914642
16, ack 1, win 58, options [nop,nop,TS val 142825 ec
r 142823], length 0
06:19:37.225266 IP (tos 0x0, ttl 64, id 12970, offse
t 0, flags [DF], proto TCP (6), length 52)
    10.10.10.101.60008 > 10.10.10.105.5001: Flags [.
], cksum 0x2908 (incorrect -> 0x39bf), ack 2, win 58
, options [nop,nop,TS val 142832 ecr 142832], length
 0
```

Figure 7.4: Traffic using different paths with the same group rule

7.7 Open vSwitch Update

The original version of Open vSwitch does not support group tables (which is included in the Mininet VM), so we had to update it to perform traffic loading. We discovered that the new version of OVS does not forward packets to the controller by default, so to get packets to the controller it is necessary to push out wildcard flow rules in a



```
 port  4: rx pkts=132211, bytes=8723832, drop=0, errs=0, frame=0, over=0, crc=0
          tx pkts=211059, bytes=12070325188, drop=0, errs=0, coll=0
          duration=207.728s
 port  3: rx pkts=149, bytes=7599, drop=0, errs=0, frame=0, over=0, crc=0
          tx pkts=93095, bytes=5354797220, drop=0, errs=0, coll=0
          duration=207.728s
```

Figure 7.5: Packets after 10 runs with iperf

proactive[7] manner at boot-up of the network. This was realized by a low-priority rule that instructs the switches to forward all types of traffic to the controller. We noted that this approach could be challenging when conducting full flushing of the network, due to the risk of removing this flow rule. Removing the rule will lead to that the switch will disregard all new incoming packets, which was originally intended to be forwarded to the controller.

7.8 Link Capacity

We found no way of gathering the link speed without creating a capacity test function at the controller to measure link each links bandwidth. In Mininet, the OVS switches ports appears to be configured to be 10 Gb, and we did not manage to change this speed. To let the controller reserve resources in forms of bandwidth, it is dependable of knowing the link bandwidth. We solved this in the simulations by generating a random bandwidth[8] and update the controller with this, followed by pushing out policing rules to the switches with the same bandwidth limit as policing threshold. This approach ensures that the there is a consistency of the controller's view of the link capacity and the actual link capacity, but it is only an approach useful for simulation purposes. Code sequence 7.3 and 7.4 show the pseudo code for applying these bandwidth limits.

Update the controllers state[9]:

Code sequence 7.3 Random generation of link bandwidth

```
linkspeed = random.randint(1,10)
link_bandwidths[link.src.dpid][link.src.port_no] = linkspeed
```

In reality, the most efficient way would likely be to monitor dropped packets since the relationship between the port speed and the actual link capacity is not always

[7] Proactive is before packets are sent/received at the switch.

[8] The validation chapter used fixed link capacities, and not random.

[9] The process is repeated for every link. link_bandwidths[link.src.dpid][link.src.port_no] represents a unique link.

Code sequence 7.4 Generation of policing limits

```
link_bw = (link_bandwidths[link.src.dpid][link.src.port_no]*1000)
sudo ovs-vsctl set Interface %s ingress_policing_rate=%s"
% (port.name,link_bw)
```

consistent. When the packets start dropping, we know that we are reaching the links maximum capacity limit, and then we could update the controller to use this capacity as the maximum limit in the resource reservation pool.

7.9 Traffic Classes

The controller divides subparts of the topology into traffic classes. The algorithm is based on multiplying the weakest link (regarding bandwidth) with the average link bandwidth on the path. The bandwidths used in the calculation are the links maximum bandwidth, not the remaining bandwidth after resource reservation.

Code sequence 7.5 Finding weighted paths

```
for path in paths:
    ....
    weighted = weakest_link*(path_bw/(len(path)-2))
    weighted_paths.append([weighted, path])
```

The algorithm executes for every possible path which the flow can travel to reach the destination. In the end, the list is sorted and divided into classes. Class 1 represents top 25%, class 3 is bottom 25%, while class 2 is the remaining. The weakest link is ergo the most determinate factor in the division of classes.

The main thoughts about introducing traffic classes are to give the administrators the ability to influence the path decision process to some extent. It can be beneficial to avoid using the lower capacity paths if the flows achieve better quality (e.g., video stream) by using more bandwidth than its absolute minimum requirement stated in the policy. By including traffic class 1 to the policy, we can be sure that the application will use the best path regarding bandwidth capacity. We can also imagine that paths with a significantly higher bandwidth than other will consist of fiber/cable instead of wireless bearers, which may imply fewer topology variances because of the static structure.

7.10 Summary of the Laboratory Work

We previously defined an assortment of requirements of which the SDN application should be able to perform. Table 7.1 summarizes how implementation solved the various requirements.

We chose to focus on bandwidth as the network property of importance in this implementation. From early on we experienced that combining policies is a very complex task, especially when conditions, actions, priorities and number of policies increases. The primary challenge is that the policies will end up affecting each other, and we need to implement smart ways of doing trade-offs which are advantageous for the affected parts. We solved this in the application by always using the policy priority as the determining factor in cases where the flow matches with multiple policies or when it is necessary to remove policies to make way for others. Policies with lower priorities will always have to give away to the higher ones. We also introduced policy checks, to ensure that an enforced policy is approved by the network before installing the policy.

We chose to develop the policy enforcement platform from scratch, and we did not include Pyretic in the controller code, which in hindsight could have eased the policy enforcement process or made it better. As of now, the approach that compiles policies into flow rules limits the communicating pairs to use one policy per flow direction. We see potential benefits of using standardized platforms to develop such a policy environment, seeing that it would remove much of the complexity for the developers, such as conveying policies to flow rules and the dispatching of the rules.

The simplified network is an approach to simulate a network with characteristics similar to the DTN. On the other hand, we believe the traffic engineering functionality of which the application conducts on the network would be beneficial use in all of the OMNs. The design might need to be changed, but some of the essential requirements are generic, and applies to particular features performed in this implementation.

Table 7.1: Implementation results

Requirement	Implementation result
Flow priority	Solved by linking flows to policies with priorities. Policies with lower priorities will always give way to higher ones. When two or more policies match with the same flow, the application chooses the policy with the highest priority. Higher priority policies can also ensure removal of flows with lower priorities to free resources.
Resource reservation	Bandwidth reservations can be made per flow. The controller keeps the state of the reservations, and maintains the requirement by placing the flow in QoS queues at the switches.
Resource utilization	The controller utilizes the network resources by allowing non-strict flows to utilize paths with filled bandwidth reservations if the average monitored traffic is low. Idle-timeout removes inactive flow rules. Hard-timeout provides freshness. Traffic loading is also contributing to the network utilization but turned out to be inoperable for providing QoS.
Randomness	The application offer randomness for path calculation and when hard-timeout is triggered. The application keeps state about the previously chosen path, whereby it is excluded from the path generation pool, ensuring choosing a new path.
Re-forwarding	The application performs re-forwarding when links are disconnected and at hard-timeout of flow rules. Re-forwarding is in practice the deletion of the particular flow rules to force the flow to be reactively be processed by the controller.
Monitoring	The controller monitors the flowing traffic on the various paths to calculate average traffic per path and per-flow. Topology knowledge is achieved by monitoring links and switches.
Traffic policing	The switches perform policing at port-level. Policing limits are set to the links bandwidth capacity and the controller proactively installs these at boot-up.
Topology Abstraction	The SDN application uses topology abstractions to filter out switches and links from its logical ”forwarding pool” if they do not support the requirements specified in the policies. After filtering, the application will try to find a path based on the remaining resources in the forwarding pool. The ARP process uses a similar approach.
Management	Management is realized by enforcing policies with an interface where the administrators can specify the policies. Limited to defining the policies before booting up the application.

Chapter 8 Discussion

This chapter is a discussion of the problems this thesis addresses. It covers a review of the results from the validation phase, where we discuss decisions, the development process, and potential improvements. A higher level discussion follows where we use the completed work to appeal to the thesis problem description.

Results from the validation show that the controller can take independent decisions which will vary according to what the policies state, combined its view of the network topology. Most importantly, this is one of the goals with traffic engineering; to adapt and to control the flows within a network based on the relationship between the topology, resources, and traffic. The developed controller application will always strive to achieve the best utilization seen from its view.

One of the goals with the validation is to present the capabilities which SDN has to offer. One of the best examples which illustrates the strength of traffic policing with SDN from the validation is shown in table 8.1. In the example, the hosts started transmitting more than the link was capable of sending. We introduced policing of the ingress traffic on the link, as well used service differentiation to police only certain parts of the traffic. This is an essential feature which the network could benefit greatly from using. By placing the high-priority military traffic in a QoS queue with higher requirements than other queues will lead to that, the policing is conducted at the other queues first.

Table 8.1: Summary of results from validation in Section 6.5

Action	**Result**
Flow 3 starts transmitting UDP traffic at 1.5 Mbit/s	0 % loss
Flow 4 starts transmitting UDP traffic at 1.0 Mbit/s	43 % loss

When the high-priority military traffic adds up, and the aggregated flows are exceeding a links capacity limitations, it is necessary to do further handling, which Table 8.2

displays. When filling the total capacity, it is of greater importance to allocate resources for the flows with the highest priority than the lower ones. The result would be the deletion and denial of the low priority flows. It is important to note that this only happens in situations where the network is reaching its maximum limitations.

Table 8.2: Summary of results from validation in Section 6.2

Action	**Result**
Flow 1 with match on policy001 in	Path 1-4 taken
Flow 2 with match on policy002 in	Removes Flow 1. Path 1-4 taken
Flow 1 enters the network again	Traffic loading using path 1-2-4 and 1-3-4

We did manage to fulfill all the requirements of the controller to some extent, and we believe that SDN delivers features which can contribute to a beneficial way of performing traffic engineering and policing in OMNs. We especially stress the controllers ability to cope with changes and variations, essentially in forms of topology changes (e.g., disconnecting links), resource changes (e.g., various link bandwidths) and traffic (e.g., flows with different needs and priorities). Since we knew on beforehand about the important features and properties of the network, we were able to take this into consideration when developing the application. We noticed the advantage by gradually being able to add features and adjusting the application as we went along, which is likely not possible when using the legacy equipment because we are dependable on the features included in the vendors software. When the controller retains topology knowledge, it emerges as the natural managing point in the network, and it also has the powers and the rights to instruct the forwarding devices to do what it wants. In contrast to legacy routing where particular management tasks can be centralized but the routers is essentially autonomous.

The design is, therefore, essential in SDN, due to new opportunities to place the control plane (controller) in various places in the network. This opens doors to a new world for network design compared to legacy network layouts which are limited by the embedded control plane in each device. It is possible to include multiple SDN controllers to the network design, and this approach would likely be required for the OMNs due to the networks characteristics and requirements. The testbed structure used a centralized approach, but we argue that in environments with extreme management link variances (such as MTNs) it can be beneficial to add more controllers to the design; moving towards a decentralized control plane approach. However, the various designs will also have different issues to solve, such as how the controllers should communicate, what information should be exchanged and when to synchronize. SDN must be designed to fit the network, followed by developing the controller(s) to fit the design.

The key element in the implementation is the use of network policies. We chose to enforce policies using a reactive approach, by which makes the flows dependent on being processed by the controller at first-entry. An advantage of this approach is that the controller gets knowledge about every flow in the network. Therefore, the controller can reserve bandwidth with certainty that a flow is in use. The stated policies are stored by the controller and are only enforced when needed to, thus is when the flow becomes active. The SDN application uses the policies as a labeling function: by matching a flow with a policy, it retains the flow's requirements. It is then up to the application to fulfill these demands, to compiling the policy action(s) into flow rules, and the dispatching of these. Due to the reactive approach, it was necessary to make the flow rules unique, so it is not possible for unprocessed flows to match with these rules. The essence of the development process was to make the linking between a user-interface where policies are defined, and the flow rules which forward the actual flows. Being based on the same principles; by using match conditions and actions, it would appear as natural to use the same match conditions in the flow rules as in the policy. This is, however, dependent on the policy enforcement design, and a reactive enforcement approach would most likely require compiling of wide policy conditions into narrow rules, such as this implementation shows.

Regarding traffic engineering in OMNs, there is a variety of inputs to adapt to, and the inputs used in the implementation are illustrated in Figure 8.1. Policies and topology are knowledge of which the SDN application obtains at boot-up of the network and enables the application to start installing particular rules before traffic is sent and received in the network. As the traffic enters the network, the controller must continuously monitor network status to detect changes in forms of traffic volume, topology events and ongoing policies, and use this as input to its traffic engineering calculations. Path decisions will, thus, depend on several factors, where the key element is performing dynamically adjustments. To monitor traffic, the controller regularly sends OF requests to the switches, and the controller performs computations based on their responses. This monitoring approach worked well in the testbed, but we acknowledge that this method does not scale well in large-scale networks, which emphasizes the importance of a robust network design when the networks grow. A network with hundreds of switches will generate a lot of management traffic by using such a request-response method.

The reactive approach lets the SDN application perform traffic engineering in a white-list manner, by ensuring that the traffic is approved before it is allowed to access the network. This is a beneficial way of preventing the network from congestion, but it is also necessary with conditional policing to cope with changes in traffic volume. OVS comes with a variety of features, of which can be utilized to achieve service differentiation, such as by combining ingress port policing and QoS queues as performed in the lab. The developed SDN application installs these features

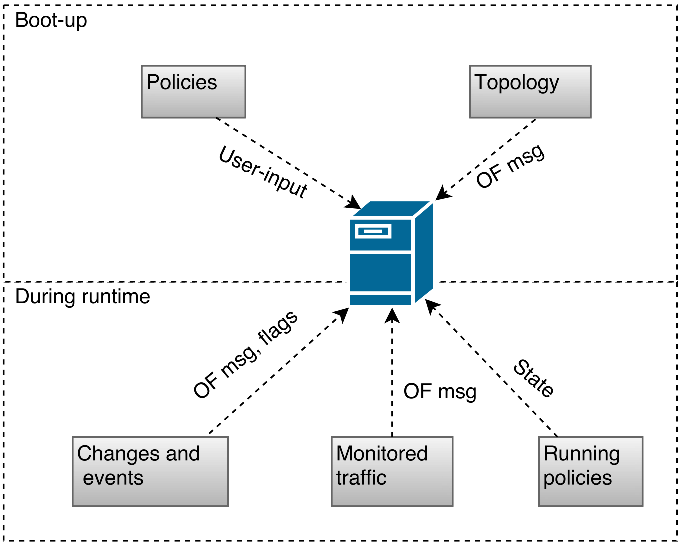


Figure 8.1: Inputs which the application adapts to

proactively based on topology knowledge gained at boot-up of the network, while placing the traffic in the various queues in a reactive manner as traffic arrives the network. The switch will start policing traffic when reaching the defined policing thresholds. The OVS switches can police traffic on port basis and also at queue basis by stating a maximum transfer limit for individual queues. Both approaches appear as robust policing methods.

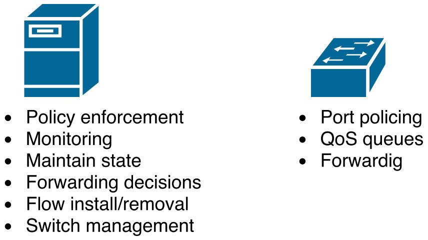


Figure 8.2: Divided responsibilities

Figure 8.2 illustrates the divided responsibilities in the SDN implementation. By inspecting the figure, we can see that all the features we have applied to the application (traffic loading, randomness, re-forwarding, bandwidth reservation, etc.) are a part of the *forwarding decisions* context. Meaning in essence, that most of the developed application is for forwarding the traffic in efficient manners. All the other things the application performs are really about obtaining inputs to be able to take the correct decisions. Due to the switches does not have the ability to make decisions, the application also manages these, such as installing policing limits and the QoS queues. Once installing these features, the application does not need to monitor or control them anymore; the responsibility is at this moment outsourced to the switches.

To police traffic at links would imply, that to some extent, there need to be policing features which the SDN switches must support. Otherwise, the controller would need to do all the job, both in forms of monitoring and the regular deletion/modification of flow rules to adjust traffic. A job that would be a tremendous for a single controller, and the approach would not scale well as the network grows. Especially considering the management traffic it would generate, which is not a good match the limited bandwidth capacities which the OMNs may have. Instead, we find it very effective to let the switches have this responsibility; the controller states the policing thresholds while the switches enforce them. We have also found that traffic engineering using the OF protocol emerges as not sufficient enough alone, being based on management of the forwarding plane and not the administration of the switch itself. In contrast, by combining OF and OVSDB, we get access to a variety of features of which emerges as adequate for performing traffic policing. Besides, we also have the possibility to include other types of management protocols into the application such as Link Layer Discovery Protocol (LLDP) and Simple Network Management Protocol (SNMP), to achieve an even greater control. The asset of an SDN application is that it is possible to include smart elements from various protocols, of which can be sewn together to make powerful functionality.

Controller processing and state correlate with the number of features, nodes, traffic and rules which the controller manages. We earlier stress the fact that flow rules should be narrowed down when installing flow rules using a reactive approach. The challenge with this strategy is the processing, the number of rules and the state this will generate as the networks expand. Especially by looking at how the developed SDN application keeps state about every running policy on each path due to maintaining a consistency of the bandwidth reservations. The application maintains a detailed state about every individual flow, and this approach does not scale well in the long run. In a large-scale network, it is necessary to cut down on the details and use wider rules and policies to keep the controllers state at an acceptable level. Instead of installing every flow reactively, a possible approach would be to enforce the policy instantly once the administrator defines it. By giving the particular policy a higher flow rule priority, we can rest assured of the flows are forwarded by this rule instead of the wider ones. However, by moving away from a reactive approach, the controller will no longer have full control of incoming traffic, which can be problematic regarding resource reservations and network utilizing. When enforcing a policy proactively with a bandwidth requirement, but no flows are using the policy on the forwarding plane, then it is a waste of reserved capacity.

Errors or misconfiguration in the SDN application may influence the whole network, and we have experienced these situations numerous of times in the laboratory implementation. The SDN application plays the most important role within the SDN network, and all forwarding devices rely on the controller, which makes the controller and the connection to the controller exposed. In contrast, the controller is also dependent on the feedback from switches to maintain consistency regarding state and view. Failures and lost data over the management links may lead to inconsistency between the controller's view of the topology and the actual topology. In the testbed, we only look at changes regarding data traffic and excluded management link variances. In a realistic SDN setting, the OF channel (management link) would probably use the same physical links as the regular traffic, thus, would suffer from the same variations. For example; a centralized SDN controller in the MTN may appear as inconvenient due to high expected management link variances. Therefore, the network should be designed and developed, so the facing challenges are mitigated in best manner. In contrast, we have documented sound capabilities of performing traffic engineering with centralized SDN approach utilizing stable management links.

To summarize, we believe that a centralized SDN solution is well suited for environments where the management links are stable, but where end-clients or dedicated traffic links rather cause the topology variations. We argue with this because the network resources are very dependent on management by the SDN application, and the application is dependent on the feedback from the switches. That said, management variances does not exclude the use of SDN, but set higher requirements for the design. Overall, SDN shows robust capabilities of forwarding traffic based on various policy conditions and offers great flexibility for the developers because of the possibility to program the control plane. In addition, flexibility also applies to the network design due to the great possibilities of placing the controller(s) on various locations in the network. This becomes particularly important when network complexity (variations, requirements, traffic volume, the number of nodes, etc.) adds up. Ergo, the SDN application can be designed and developed to fit a particular network environment, based on requirements and obvious challenges. The SDN suite includes, but is not limited to, the OF protocol, of which the application uses to control the forwarding plane on the switches. Despite this, at times, it would appear as necessary to include other protocols to the application to offer features beyond OF, such as OVSDB which we used in the testbed to enforce flow policing and QoS. We think that it is beneficial to outsource some features to the switches, in a way that keeps the generated management traffic and controller processing to the minimum.

Overall, we are satisfied with our implementation and the carried out results. We are, however, aware of that the application developed in this thesis has many imperfections and is still far from being a complete solution. The implementation is proof that we can conduct traffic engineering and policing by using SDN in a simplified and simulated military network environment, whereby using the experience gained, and results as a basis for the discussion on the topic.

Chapter 9 Conclusion

In this thesis, we have shown that Software Defined Networking can be used for traffic engineering and policing in an Operational Military Network. The problem this thesis addressed was to look at how SDN can be utilized as a tool for traffic engineering and policing in an OMN. OMNs are complex networks due to the high network requirements combined with significant exposure to resource limitations and variances.

This thesis describes requirements and challenges which the different OMNs are facing, and we point out beneficial solving of these with the use of SDN. The various OMNs have different characteristics and structure, and we have found that these factors have a great impact in how the SDN architecture should be designed. This thesis addresses the management links and the controller processing as the major determining design factors.

Enforcing policies is a way of performing traffic engineering, and we propose designs for how policy enforcement can be conducted using an SDN framework, whereby choosing a custom policy approach for the laboratory implementation.

Based on our findings, we have designed and developed an SDN application which manages a simulated OMN. The application has been validated in scenarios where policies, topology, traffic and resources vary. We have shown by utilizing SDN capabilities, that the controller can monitor the network, and use this input together with predefined policies to prioritize, police, ensure QoS, and dynamically adjust the flows. We find it beneficial to let the switches police traffic on port or queue basis.

Based on findings during the work with this thesis, we conclude that SDN for traffic control and policing in networks exposed to limitations and traffic link variations, such as in OMNs, appears as a valuable approach due to offered capabilities and flexibility in the design and development process. However, SDN does not solve every network problem and face many challenges in the same way as traditional networks.

9.1 Future Work

Due to the SDN overlay, management traffic may use the same links as the regular traffic. Following this, we have found that more research is beneficial on this topic to identify if SDN is still a solid approach by taking management traffic variances in consideration. It may also be helpful to document other approaches of performing policy enforcement using SDN, such as third-party platforms as Pyretic.

List of topics which could have been of interest to research:

– Implement an Operational Military Network with SDN where management links are suffering from variances. Test various approaches to controller design within the network.

– Implement an Operational Military Network by including Pyretic to the applications code to explore the advantages of this policy approach.

References

[1] Mohammed M Alani. Tcp/ip model. In *Guide to OSI and TCP/IP models*, pages 19–50. Springer, 2014.

[2] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao. Overview and principles of internet traffic engineering. RFC 3272, RFC Editor, May 2002.

[3] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. RFC 2475, RFC Editor, December 1998. http://www.rfc-editor.org/rfc/rfc2475.txt.

[4] Rolv Bræk. SDL basics. *Computer Networks and ISDN Systems*, 28(12):1585–1602, 1996.

[5] H.W. Braun. Models of policy based routing. RFC 1104, RFC Editor, June 1989.

[6] Marco Cello. Software defined networking (sdn). Talk at IEIIT – Consiglio Nazionale delle Ricerche, March 2014.

[7] Cisco. Understanding rapid spanning tree protocol (802.1w). Technical Report 24062, October 2006.

[8] T. Clausen and P. Jacquet. Optimized link state routing protocol (olsr). RFC 3626, RFC Editor, October 2003. http://www.rfc-editor.org/rfc/rfc3626.txt.

[9] Ryu SDN Framework Community. What's ryu? http://osrg.github.io/ryu/. Accessed: 2016-04-25.

[10] David Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013.

[11] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2), 2014.

[12] Project Floodlight. Floodlight. http://www.projectfloodlight.org/floodlight/. Accessed: 2016-05-16.

[13] Flowgrammable. Flowmod. http://flowgrammable.org/sdn/openflow/message-layer/flowmod/. Accessed: 2016-04-19.

[14] Flowgrammable. Hello. http://flowgrammable.org/sdn/openflow/message-layer/hello/. Accessed: 2016-04-20.

[15] Open Networking Foundation. Openflow switch specification version 1.3.0. 2012.

[16] Open Networking Foundation. Software-defined networking: The new norm for networks. White paper, Open Networking Foundation, 2012.

[17] Paul Goransson and Chuck Black. *Software Defined Networks: A Comprehensive Approach*. Elsevier, 2014.

[18] E. Haleplidis, K. Pentikousis, S. Denazis, J. Hadi Salim, D. Meyer, and O. Koufopavlou. Software-defined networking (sdn): Layers and architecture terminology. RFC 7426, RFC Editor, January 2015. http://www.rfc-editor.org/rfc/rfc7426.txt.

[19] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43. ACM, 2013.

[20] Java. About. https://java.com/en/about/. Accessed: 2016-04-21.

[21] Jaxenter. The SDN evolution: Out-of-band is where it's at. https://jaxenter.com/the-software-defined-networking-evolution-out-of-band-is-where-its-at-121834.html. Accessed: 2016-04-20.

[22] Diego Kreutz, Fernando MV Ramos, P Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, 103(1), 2015.

[23] Madhusanka Liyanage, Andrei Gurtov, and Mika Ylianttila. *Software Defined Mobile Networks (SDMN): Beyond LTE Network Architecture*. John Wiley & Sons, 2015.

[24] The Newstack. SDN Series Part Four: Ryu, a Rich-Featured Open Source SDN Controller Supported by NTT Labs. http://thenewstack.io/sdn-series-part-iv-ryu-a-rich-featured-open-source-sdn-controller-supported-by-ntt-labs/. Accessed: 2016-04-25.

[25] ONOS. About. http://onosproject.org/. Accessed: 2016-05-16.

[26] OpenDaylight. https://www.opendaylight.org/. Accessed: 2016-04-26.

[27] B. Pfaff and B. Davie. The open vswitch database management protocol. RFC 7047, RFC Editor, December 2013. http://www.rfc-editor.org/rfc/rfc7047.txt.

[28] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular SDN programming with Pyretic. *Technical Report of USENIX*, 2013.

[29] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, David Walker, and Princeton Cornell. Composing software defined networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Citeseer, 2013.

[30] Hans Fredrik Skappel. Software defined networking for policy enforcement in military networks. Specialization project, Norwegian University of Science and Technology, December 2015.

[31] Rune Linchausen Skar. Systemdynamisk tilnærming for risikoanalyse av transformasjonen til nettverksbasert forsvar. pages 62–64, 2006.

[32] Gary N Stone, Bert Lundy, and Geoffrey G Xie. Network policy languages: a survey and a new approach. *Network, IEEE*, 15(1), 2001.

[33] John Strassner and E Ellesson. Terminology for describing network policy and services. *draft-strassner-policy-terms-02. txt*, 1999.

[34] Erik Sørensen. Evaluating software defined networking for use in military networks. Specialization project, Norwegian University of Science and Technology, January 2014.

[35] Erik Sørensen. Sdn used for policy enforcement in a federated military network. Master thesis, Norwegian University of Science and Technology, June 2014.

[36] Mininet Team. Introduction to mininet. https://github.com/mininet/mininet/wiki/Introduction-to-Mininet. Accessed: 2016-06-04.

[37] Olivier Tilmans and Stefano Vissicchio. Igp-as-a-backup for robust sdn networks. In *Network and Service Management (CNSM), 2014 10th International Conference on*, pages 127–135. IEEE, 2014.

[38] D Vassis, A Tsakrikadakis, K Panagiotopoulos, G Kormentzas, D Vergados, and F Lazarakis. Building robust military networks using advanced software tools. 2002.

[39] Open vSwich. Rate-limiting vm traffic using qos policing. http://openvswitch.org/support/config-cookbooks/qos-rate-limiting/. Accessed: 2016-06-12.

[40] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Terminology for policy-based management. RFC 3198, RFC Editor, November 2001.

[41] Øivind Kure and Ingvild Sorteberg. Network architecture for network centric warfare operations. FFI Report 01561, 2004.

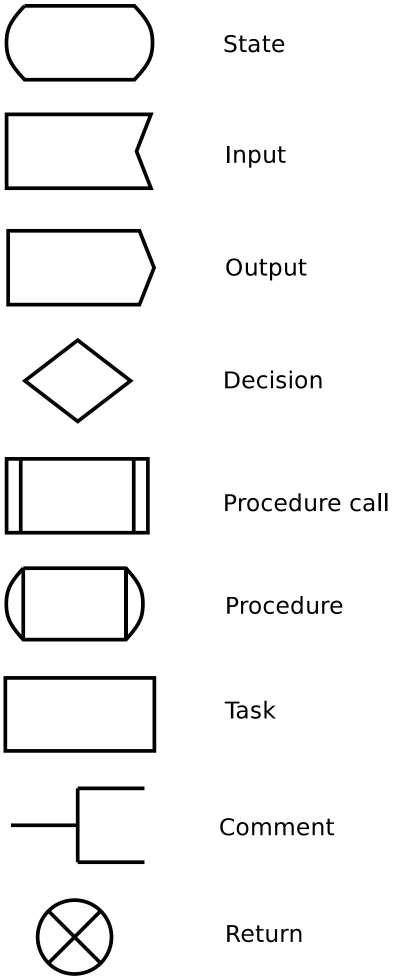
Appendix A

Mininet Topology Source Code

```python
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call

def myNetwork():

    net = Mininet( topo=None,
                   build=False,
                   ipBase='10.0.0.0/8')

    info( '*** Adding controller\n' )
    c0=net.addController(name='c0',
                      controller=RemoteController,
                      ip='10.10.10.126',
                      protocol='tcp',
                      port=6633)

    info( '*** Add switches\n')
    s3 = net.addSwitch('s3', cls=OVSKernelSwitch, dpid='3')
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch, dpid='2')
    s4 = net.addSwitch('s4', cls=OVSKernelSwitch, dpid='4')
    s1 = net.addSwitch('s1', cls=OVSKernelSwitch, dpid='1')

    info( '*** Add hosts\n')
    h5 = net.addHost('h5', cls=Host, ip='10.10.10.105',
    ↪   defaultRoute=None)
    h8 = net.addHost('h8', cls=Host, ip='10.10.10.108',
    ↪   defaultRoute=None)
    h4 = net.addHost('h4', cls=Host, ip='10.10.10.104',
    ↪   defaultRoute=None)
    h3 = net.addHost('h3', cls=Host, ip='10.10.10.103',
    ↪   defaultRoute=None)
    h1 = net.addHost('h1', cls=Host, ip='10.10.10.101',
    ↪   defaultRoute=None)
    h6 = net.addHost('h6', cls=Host, ip='10.10.10.106',
    ↪   defaultRoute=None)
    h2 = net.addHost('h2', cls=Host, ip='10.10.10.102',
    ↪   defaultRoute=None)
    h7 = net.addHost('h7', cls=Host, ip='10.10.10.107',
    ↪   defaultRoute=None)

    info( '*** Add links\n')
    net.addLink(s4, s2)
    net.addLink(h1, s1)
    net.addLink(h2, s1)
    net.addLink(s1, s3)
    net.addLink(s1, s2)
    net.addLink(s3, s4)
    net.addLink(s1, s4)
    net.addLink(s4, h5)
    net.addLink(s4, h6)
    net.addLink(s4, h7)
    net.addLink(s4, h8)
    net.addLink(h3, s1)
    net.addLink(h4, s1)

    info( '*** Starting network\n')
    net.build()
    info( '*** Starting controllers\n')
    for controller in net.controllers:
        controller.start()

    info( '*** Starting switches\n')
    net.get('s3').start([c0])
    net.get('s2').start([c0])
    net.get('s4').start([c0])
    net.get('s1').start([c0])

    info( '*** Post configure switches and hosts\n')
    s3.cmd('ifconfig s3 10.10.10.3')
    s2.cmd('ifconfig s2 10.10.10.2')
    s4.cmd('ifconfig s4 10.10.10.4')
    s1.cmd('ifconfig s1 10.10.10.1')

    CLI(net)
    net.stop()

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

Appendix B

SDL Symbols


State
Input
Output
Decision
Procedure call
Procedure
Task
Comment
Return