NTNU
Norwegian University of
Science and Technology

SDN in Heterogeneous Mobile Tactical Networks

Håvard Magne Fagervoll

Master of Science in Communication Technology
Submission date: June 2017
Supervisor: Øivind Kure, IIK
Co-supervisor: Ragnar Wik, Kongsberg

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title: SDN in Heterogeneous Mobile Tactical Networks

Student: Håvard Magne Fagervoll

Problem description:

Software Defined Networking is an approach to networking where the control plane is decoupled from the data plane. It has been well received by the network community and praised as the next generation in networking. The approach allows for programmable networks designed for its specific purpose. Military networks are currently developing fast, and may now consist of numerous unmanned entities deployed in a network to solve critical missions. They each have their purpose in the network and data to deliver to other units or a controlling point in the network. As the mission develops, the requirements for the individual data streams changes. The location of an entity may have to be broadcasted to all other entities or a video stream may currently be critical for a mission, while later it should just offered as a service and not prioritized throughout the network. To offer a more robust network, an additional radio interface could be added with a tradeoff of lower bandwidth but longer range. The network can now be considered a heterogeneous network. However this imposes higher complexity on the network.

Objective: The objective of this thesis is to explore how the concept of Software Defined Networking can be used to address the problems occurring for the suggested infrastructure consisting of a network of multi radio unmanned entities.

Methodology: The candidate needs to identify what limitations the circumstances put on an approach using software defined networking. Also how it can be incorporated into the suggested infrastructure, both physically and logically. The approaches should be thoroughly discussed. A proof of concept should be implemented for one of the approaches.

Responsible professor: Øivind Kure, ITEM

Supervisor: Ragnar Wik, Kongsberg

Abstract

Military operations are increasingly dependent on networks and transferring of data. A heterogeneous mobile tactical network, is a complex network with dynamic characteristics used during military operations. In the field, resources are scarce, and the network can quickly get congested. To optimize the network to efficiently route traffic and handle prioritized data, a method offering fine grained control of the network would be beneficial. Software Defined Networking (SDN) has proven to facilitate granular control for several types of networks. This thesis looks at how Software Defined Networking (SDN) can be incorporated and used in a heterogeneous mobile tactical network. It states that the first step towards an SDN network is to build topology for the network. Three conceptual models are discussed, and an approach where topology is collected from a local legacy router is implemented.

Sammendrag

Militære operasjoner er i økende grad avhengig av nettverk og overføring av data. Et heterogent mobilt taktisk nett, er et komplekst nettverk med dynamiske egenskaper i bruk under militære operasjoner. I felten er det begrensede ressurser, og nettverket kan raskt bli overbelastet. For å optimalisere nettverket for å lede trafikk effektivt og håndtere prioritering av data, vil det være fordelaktig å benytte en metode som tilbyr detaljert forvaltning av nettverket. Software Defined Networking (SDN) har vist seg å fasilitere finkornet styring av ulike typer nettverk. Denne oppgaven vurderer hvordan SDN kan bli integrert and benyttet i et heterogent mobilt taktisk nettverk. Det blir konstatert at første steg for en overgang til et SDN-nettverk, er å bygge topologi for nettverket. Tre konseptuelle modeller blir diskutert, og en tilnærming hvor topologi blir hentet fra en lokal tradisjonell router er implementert.

Preface

This thesis constitutes the fulfillment of my Masters of Science degree at the Department of Telematics at Norwegian University of Science and Technology (NTNU) in Trondheim, Norway.

I would like to thank Øivind Kure for providing support and guiding me throughout this thesis. I would also like to thank Ragnar Wik and Anne Marie Hegland for providing support and feedback.

My friends at the Paradise Office, also deserve to be thanked for swift feedback and creative solutions for some of the problems encountered.

I would also like to thank my Mum and Dad, my brother Steinar, and Marte. For helping out when they could.

Contents

List of Figures **ix**

List of Tables **xi**

List of Acronyms **xiii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 What this thesis will be about and how it is carried out (Methodology) 2
1.3 Structure of the document . . . 2

2 Theory **3**
2.1 Software Defined Networking . . . 3
2.1.1 OpenFlow . . . 5
2.2 Mobile Ad hoc Network . . . 8
2.3 Mobile Tactical Networks . . . 9
2.4 Heterogeneous Networks . . . 11

3 SDN in heterogeneous mobile tactical networks **13**
3.1 Scenario . . . 13
3.2 Heterogeneous Capabilities . . . 14
3.3 SDN . . . 16

4 Architecture design **19**
4.1 Proposed models . . . 19
4.1.1 Model 1 . . . 19
4.1.2 Model 2 . . . 22
4.1.3 Model 3 . . . 25

5 Implementation **27**
5.0.1 Available controllers . . . 27
5.1 Ryu . . . 29
5.2 Mininet . . . 30

5.3 Implementation 31
5.3.1 Topology 31
5.3.2 Topology Parser 33

6 Experiences from the Implementation **37**

7 Discussion and Conclusion **39**
7.1 Discussion 39
7.2 Conclusion 40
7.3 Future Work 40

References **41**

Appendices

A Mininet Topology Script **43**

List of Figures

2.1 Sample topology for an SDN network. Each host is connected an SDN switch. The SDNs switches are all controlled by a centralized controller. 4
2.2 Layered SDN architecture. From [21] 5
2.3 Simplified view of how OpenFlow (OF) handles incoming packets. . . . 7
2.4 A Three-level network topology, from [16]. This figure illustrates a scenario where the strategic backbone residing in Norway, is connected to a deployable tactical network. The deployable network aggregates traffic from several locations. 10
2.5 Example of integration of wireless access networks. The User Equipment (UE) has access to both the Wireless Local Area Network (WLAN) and the cellular network. 12

3.1 Illustration of a mobile tactical network with a variety of nodes. Command & Control (C2) is the commanding entity, node 1 and 2 are ground nodes, node 3 is an aerial node, and node 4 is a surface node. 14
3.2 The proposed architecture for a platform consisting of 2 Radio Access Technologys (RATs), a local system, and a switch interconnecting them. 15
3.3 Example topology for the platforms illustrated in figure 3.2. Radio 1 (white) are all interconnected, while Radio 2 (grey) only have links from node 1 to 2 and from node 2 to 3. 16

4.1 Platform design with one SDN switch interconnecting the radios and the local system. 20
4.2 Platform design with one SDN switch in each of the radios. 23
4.3 Platform design with SDN switch in only one radio. 25
4.4 Platform design where the routers in the radios are substituted by SDN switches. 25

5.1 Ryu architecture. From [20]. The controller is running in the control layer, and the SDN applications are running in the application layer. . . 29
5.2 Topology of the network implemented using Mininet. 32

List of Tables

2.1 Overview of the fields in a flow entry in the flow table. 6

3.1 Comparison of the available communication links and their characteristics. Radio 1 offers low throughput, but long range. Radio 2 provides higher throughput, but shorter range. 15

5.1 Comparison of the available open source SDN controllers. 28
5.2 Table for IP version 4 (IPv4) addresses for the interfaces of the routers in the implemented network. 31

List of Acronyms

API Application Programming Interface.

BGP Border Gateway Protocol.

C2 Command & Control.

CLI Command Line Interface.

GUI Graphical User Interface.

HetNet Heterogeneous Network.

IDS Intrusion Detection System.

IETF Internet Engineering Task Force.

IP Internet Protocol.

IPv4 IP version 4.

IPv6 IP version 6.

JSON JavaScript Object Notation.

MANET Mobile Ad hoc Network.

MPLS Mutliprotocol Label Switching.

NEM Network Equipment Manufacturer.

NFV Network Function Virtualization.

NTNU Norwegian University of Science and Technology.

NTT Nippon Telegraph and Telephone.

OF OpenFlow.

ONF Open Network Foundation.

OS Operating System.

OSPF Open Shortest Path First.

OVS Open Virtual Switch.

OXM OpenFlow Extensible Match.

QoS Quality of Service.

RAT Radio Access Technology.

REST Representational state transfer.

RFC Request for Comment.

RSSI Received Signal Strength Indication.

SDN Software Defined Networking.

SNMP Simple Network Management Protocol.

SNR Signal-to-noise ratio.

TC Traffic Control.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TLV Type-Length-Value.

UE User Equipment.

VLAN Virtual LAN.

WLAN Wireless Local Area Network.

XML eXtensible Markup Language.

Chapter 1
Introduction

This chapter first explains some of the motivation for this thesis. Then a summary of what the thesis will be about, and lastly the structure of the document is described.

1.1 Motivation

A modern military network consists of multiple mobile tactical units deployed to perform predetermined mission critical tasks. Each node has its assets, drawbacks and purpose in the network, but to perform the mission it is destined for, it is necessary to communicate with a controlling entity. The network is interconnected via a radio technology employing a multihop routing protocol. The radio utilizes a shared medium and has certain characteristics like range, bandwidth and delay associated with it. As the mobile units move, these characteristics change and the link may even perish when moving too far.

By extending the network with one additional radio access technology with the tradeoff of lower throughput for longer range, more flexibility is offered to the system. Each node will then be connected to two separate radio interfaces- Now there are even more routes for the data to take to arrive at the its destination. As the new access technology is added to the system, it forms an even more complex system with new potential, challenges and limitations. It is now a heterogeneous network.

SDN has been suggested to facilitate complex network in need of strict policy management for optimization of the available bandwidth. SDN is an approach to networking where the control plane is decoupled from the data plane. The control layer is moved to a centralized intelligence where the data plane is abstracted for network applications running on top of the control layer. The network applications are offered granular control mechanisms for the data flows throughout the network.

1.2 What this thesis will be about and how it is carried out (Methodology)

This thesis will look at how SDN can build topology in a heterogeneous military tactical network with a hybrid structure consisting of both legacy network components and SDN entities. The complexities of a scenario encountered by Kongsberg, are explained and structured. Three conceptually different architecture models are proposed and discussed. Lastly one of the approaches for Model 1 is implemented as a proof of concept

1.3 Structure of the document

Chapter 2 presents and explains a selection of terms used throughout the thesis. Next, chapter 3 explains the complexities and limitations of the scenario. After the problem is described, chapter 4 proposes several approaches for the architecture design of the platforms. Then, chapter 5 implements one of the proposals from chapter 4. In the end, experiences from the implementation is presented in chapter 6 and chapter 7 has a discussion and the conlusion for the thesis.

Chapter 2
Theory

This chapter will present the concepts encountered and discussed throughout this thesis. It will establish common grounds before heading into the scenarios, evaluations and models presented later. First SDN will be introduced along with its accompanying protocol OF. Next, Mobile Ad hoc Network (MANET) will be introduced, followed by the mobile tactical network. Lastly, Heterogeneous Networks (HetNets) will be introduced.

2.1 Software Defined Networking

SDN is an approach to networking where the data plane is decoupled from the control plane. In legacy networks, the control for each device is distributed and assigned to each individual device. The concept of SDN decouples this control plane and moves it to a centralized software intelligence called a controller. The network devices are then programmatically controlled and configured by the controller. Figure 2.1 illustrates a sample topology for an SDN network and how the controller is connected to the network devices.

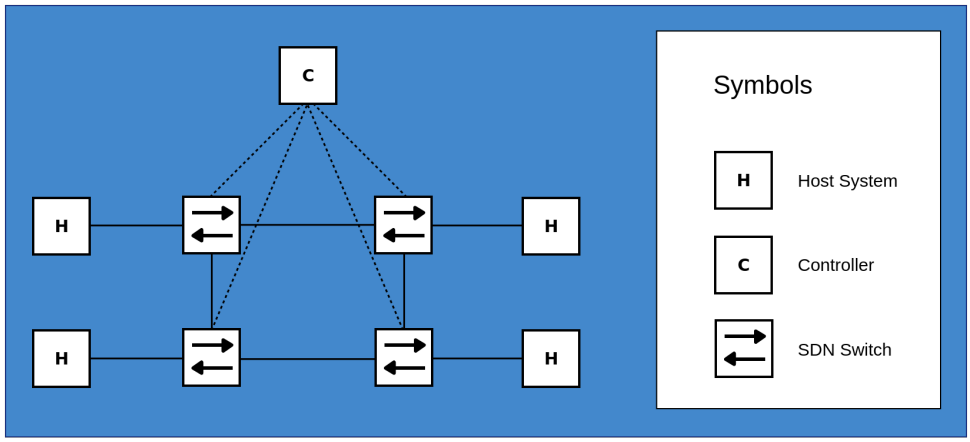

Figure 2.1: Sample topology for an SDN network. Each host is connected an SDN switch. The SDNs switches are all controlled by a centralized controller.

Figure 2.2 illustrates the layered architecture of SDN, where the infrastructure layer represents the data plane, also referred to as the forwarding plane. The infrastructure layer comprises the physical devices, which is controlled by the upper layers. The control layer communicates with the infrastructure layer via a south-bound Application Programming Interface (API), and abstracts the underlying network, providing a simplified view of the network to the application layer. The application layer builds on top of the control layer and can via the north-bound API offered by the controller, create complex networking applications.

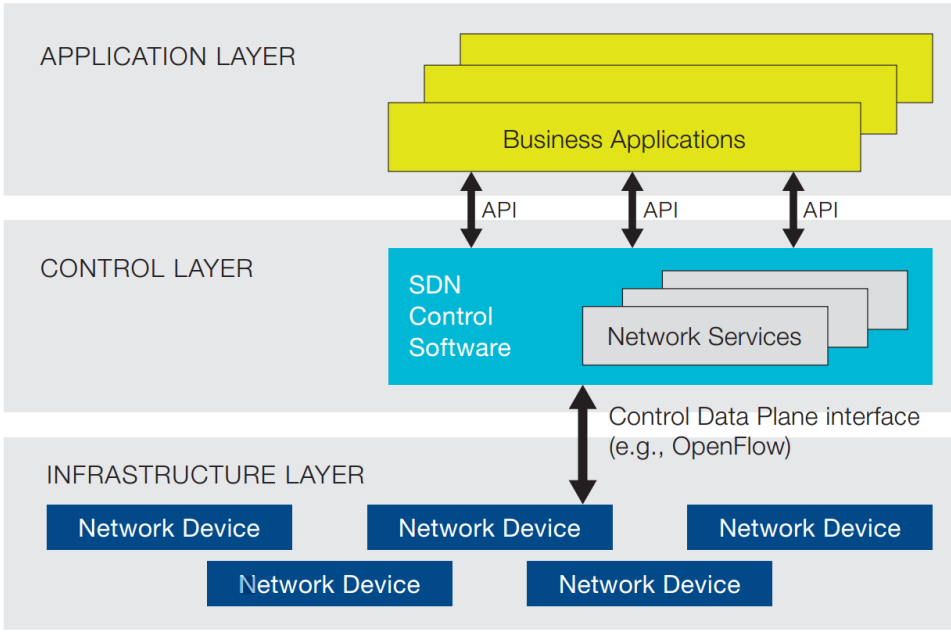

Figure 2.2: Layered SDN architecture. From [21]

The term controller is often used for the combined functionality of both the control and the application layer. A controller maintains a view over the entire network, and implements policies regarding forwarding, priorities, and load balancing. The controller often comes bundled with a range of business applications that provide these functions using the north-bound API, often implemented in Python, Java or Representational state transfer (REST), provided by the control layer. A standard for the north-bound API has yet to be defined and acknowledged.

2.1.1 OpenFlow

The OpenFlow (OF) protocol is an open standard that was initially developed to facilitate network research and development, but when published in [19], it quickly caught the interest of research communities and network vendors. In 2011 the development was moved to the newly founded Open Network Foundation (ONF), which would keep developing it with a strong philosophy for open source. OF is now the most widely used south-bound API, and it is the only defined SDN standard [23].

OF utilizes the existing forwarding tables in network devices, earlier populated by the distributed routing algorithms. OF defines this as a flow table consisting of

flow entries. The flow entries are set using the OF protocol, which provides a set of messages to program the switches. A flow is *"a set of packets transferred from one network endpoint (or set of endpoints) to another endpoint (or set of endpoints). The endpoints may be defined as IP address-TCP/UDP port pairs, VLAN endpoints, layer three tunnel endpoints, or input ports, among other things"*[12].

Match Field	**Priority**	**Counters**	**Instructions**	**Timeouts**	**Cookie**	**Flags**

Table 2.1: Overview of the fields in a flow entry in the flow table.

Flow entries are used to match and process packets, and is illustrated in 2.1. A flow entry contains one or more match fields for matching the packets. The fields available for matching includes all fields in Ethernet, Virtual LAN (VLAN), Mutliprotocol Label Switching (MPLS), IPv4 and IP version 6 (IPv6). It also offers OpenFlow Extensible Match (OXM), which describes Type-Length-Value (TLV) pairs that can be used to define any header field. As an example the rule could match with all packets with a specified IPv4 address. The priority field, gives the entry precedence over other entries. Counters are used to gather statistics for the flows. This is typically counting packets and bytes. The instructions field contains actions to execute for incoming packets. The packet may be forwarded, dropped or passed to the controller. It can also execute more advanced features before passing it on, such as editing the header fields. If the flow entry is idle for longer than the timeout threshold, the flow is removed. The cookie is a value chosen by the controller, and does not affect the packet processing. Flags offer additional life cycle functionality to the flows. It can be set to notify the controller when the flow times out [22]. The flow rules configured by the controller and the outcome of these rules are one of the fundamental actions:

- Forward
- Drop
- Pass to the controller

Incoming packets are first matched against the Flow Tables, as illustrated in figure 2.3. The packets can trigger a flow entry with actions to execute. It can instruct the switch to forward the packet on a physical or logical port. The packet will then be dispatched out on the network. Or the packet can match a flow rule defining a filtering scheme which drops the packets. Packets that require to be processed by the controller, will be consumed and dispatched to the controller. The controller will respond to the packet, for example by installing new flow entries. The

communication between the SDN switch and the controller is performed over the OF protocol. The most basic controller defines, adds and deletes flow entries.

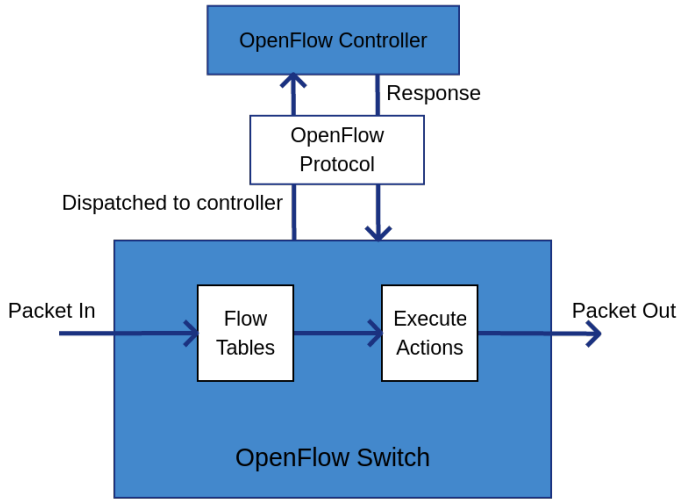

Figure 2.3: Simplified view of how OF handles incoming packets.

Flow entries can be installed in reactively, where it passes packets not matching any of the flow entries, to the controller. The controller can then determine how the flow should be treated and install the appropriate flow entries. The scheme can also be to act pro-actively, installing rules for the expected flows. More general flow entries can be used to catch those not matching with other entries. This is up to the controller, and the apps running on top of it.

The switch is usually connected to the controller over a Transport Layer Security (TLS) encrypted channel, but may also be over Transmission Control Protocol (TCP). There are two conceptual ways of connecting the SDN device to the controller, in-band and out-of-band. In the latter case, the controller is connected outside of the OF pipeline. This requires basic Internet Protocol (IP)/TCP networking. For in-band connections, the switch may allow the controller to initiate the connection, or else, the switch has to be pre-configured with the proper flow entries.

An OF enabled device can be considered a layerless device, as only the controller and the application running on top, decide what layer the device is interpreting. However, the terms SDN Switch or OF Switch has been coined as devices supporting OF and will be used intermittently throughout this thesis.

An SDN controller initially builds topology by registering switches when they

connect to the controller. Then further topology discovery is up to the application running on top of the controller, for example a layer 2 learning switch.

2.2 Mobile Ad hoc Network

As IP networks advanced from conventional wired networks into more advanced wireless ones, the community realized the need to establish a body to research and develop standards for the fast developing dynamic networks. The MANET working group was formed by the Internet Engineering Task Force (IETF), and quickly released a Request for Comment (RFC) [18] in 1999. This RFC tried to describe and explore what they defined as a MANET. The MANET working group has now defined their purpose:

"*The purpose of the MANET working group is to standardize IP routing protocol functionality suitable for wireless routing applications within both static and dynamic topologies with increased dynamics due to node motion or other factors.*"[3]

A MANET can be described by having mobile nodes with wireless access technology and routing capabilities. These nodes typically form an ad hoc, multihop stub network with few or no connections to other networks. The nodes in the network will provide the necessary routing functionality to efficiently route traffic as needed. Reference [18] describes MANETs in this nonchalant way:

"*It is, simply put, improved IP-based networking technology for dynamic, autonomous wireless networks.*"[18]

MANETs now encompass a wide range of networks with common, typically fluctuating, characteristics. Reference [18] presents a list of some of the prominent characteristics, which is presented in short form below.

- **Dynamic topology**. The topology of the network is usually dynamic, as the nodes are free to geographically move around.
- **Bandwidth-constrained**. The wireless technologies deployed on these nodes often offer limited bandwidth. As the nodes move, the characteristics of the links will also change. This characteristic also often leads to congestion throughout the network, and thus traffic policing emerge as an important component.
- **Energy-constrained**. Mobile nodes carried by systems powered by limited power sources, have to preserve energy.
- **Limited physical security**. Wireless networks are in general more prone to attacks than wired networks.

The relevant characteristics varies for different types of networks and use cases. Relevant for this thesis are the MANETs used for military purposes.

2.3 Mobile Tactical Networks

Reference [16] proposes a taxonomy for the network architecture of a military network. The proposed network is divided into three layers; backbone, aggregation and mobile tactical networks. The backbone network has similarities to the core network layer of the Cisco hierarchical network model, which interconnects different sites and areas of an enterprise. This is called the strategic backbone network, and it is a national network connected to headquarters and operational bases around the world. This is illustrated in figure 2.4 connecting the Deployable tactical network to the strategic backbone network in Norway. The aggregation layer comprises a deployable tactical network and a tactical backbone, both offering connectivity to the strategic backbone. Deployable tactical networks are networks residing where operations are currently ongoing, domestic or abroad. They are usually connected to the backbone via tunnels or satellite links. The strategic backbone network is similar to the deployable tactical network, but should offer connectivity for mobile tactical networks. This means it has to be equipped with all the necessary radio technology to communicate with the appropriate forces and networks.

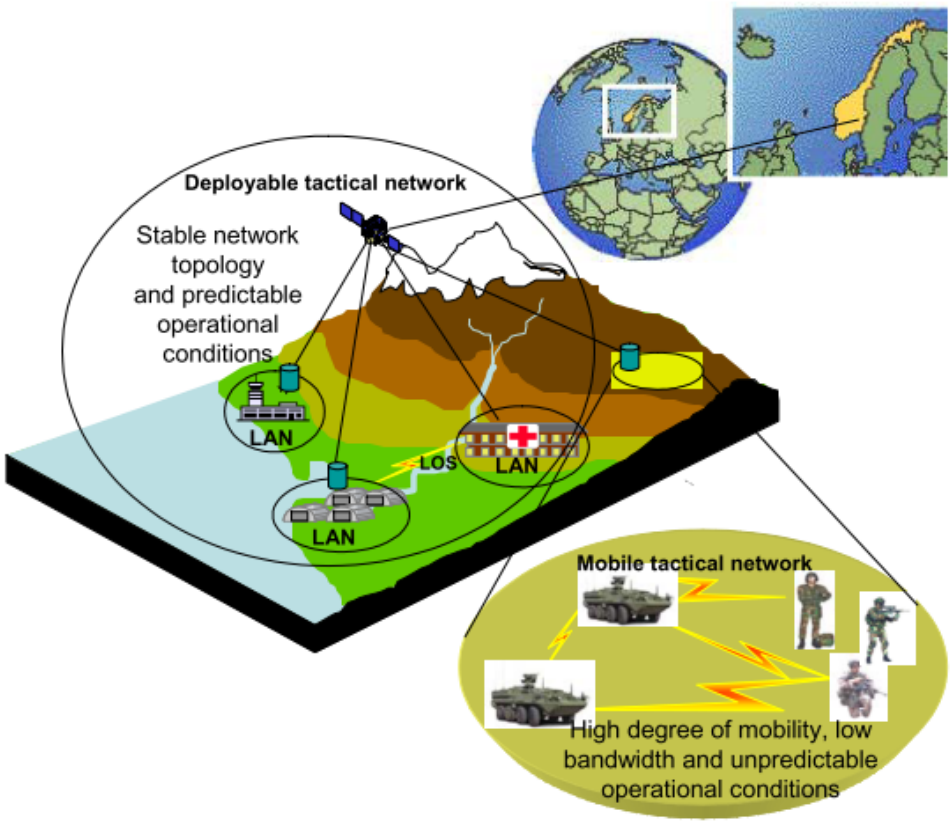

Figure 2.4: A Three-level network topology, from [16]. This figure illustrates a scenario where the strategic backbone residing in Norway, is connected to a deployable tactical network. The deployable network aggregates traffic from several locations.

The mobile tactical network is a MANET deployed during an operation or mission. Due to the nature of military operations, these networks are very different from commercial networks. The network consist of mobile units, that being manned or unmanned, aerial-, ground- or sea vehicles, or individual soldiers. These nodes, while in motion, changes the topology in a rapid pace. The nodes, connected using military grade radio interfaces with possible scarce resources, will experience big variations in connectivity, link capacity and network topology. A mobile tactical network has to be formed in this way, because during military operations, it can not rely on public or static infrastructure. Static infrastructure will not be feasible where the operations are being carried out, both because of the mobile nature of an operation, but also because of geographical location. It can not rely on public infrastructure either, as it

is prone to attacks and may be compromised or disconnected.

2.4 Heterogeneous Networks

HetNets are used to describe a network consisting of different types of equipment implementing inequal software. Software include communication protocols, implementations and configurations. The various types of equipment can also have varying channel characteristics, including range and frequency.

Wireless access technologies allow peers to communicate using the air between them. By using parts of the frequency spectrum as the channel for communication, data can be transmitted and received while roaming. Different wireless technologies has been developed and seen wide deployment, in particular for WLANs and cellular networks. Each of the technologies have their particular characteristics derived from factors like frequency, protocols and purpose. For example WLANs have relatively short range, but offer high throughput, while cellular networks offer longer range, but lower throughput.

As these technologies have matured, it has become evident that some of these technologies complement each other in certain scenarios. Together they can for instance satisfy both the need of roaming in a larger areas, as well as offering high bandwidth when remaining close to key locations. When such a network is realized, and two or more RATs are integrated, it is referred to as a HetNet. Figure 2.5 shows an example of how HetNets are deployed and utilized. The figure shows some UE, illustrated as a cellphone, capable of connecting to both a WLAN and a cellular network.

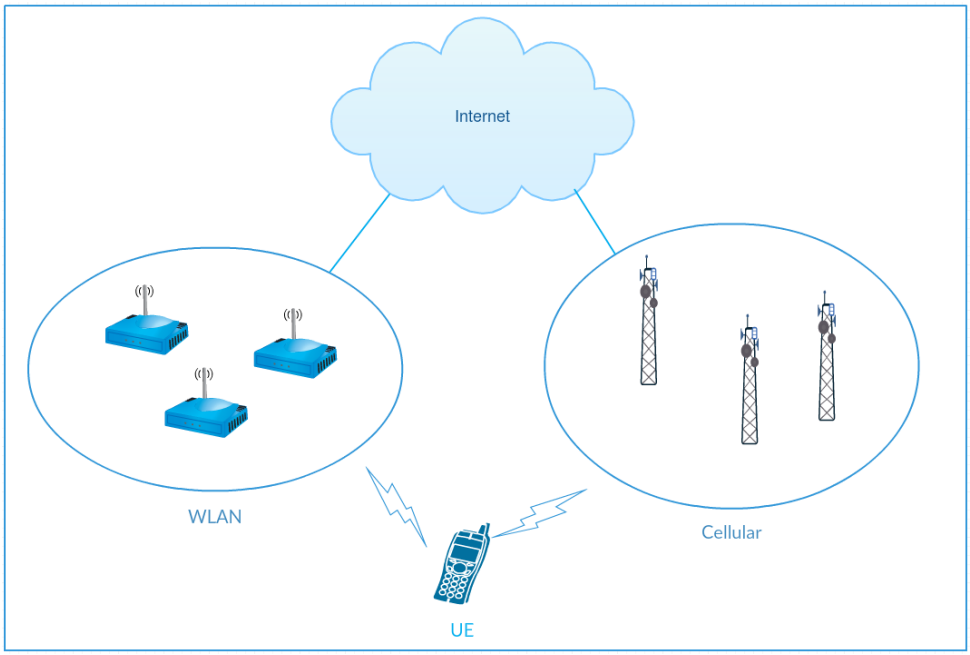

Figure 2.5: Example of integration of wireless access networks. The UE has access to both the WLAN and the cellular network.

The selection of which RAT to use when, remains a problem area where extensive research has been done [14]. Typically a decision of what RAT to use is made when a new network arrives [13]. This decision has been characterized as an optimization issue [15], where the goal is to distribute the traffic over several RATs to maximize the network flow. Other HetNets are integrated with other intentions. What they have in common is that the end-user should be agnostic to the decision. It should be handled by the network.

The reader should understand that a HetNet is deployed to satisfy one or more Quality of Service (QoS) parameters. For example to offer:

- Throughput
- Robustness
- Priority
- Availability

Chapter 3
SDN in heterogeneous mobile tactical networks

This chapter will present the scenario encountered by Kongsberg. The nodes in the network are defined and explained. HetNet capabilities is added to the scenario before SDN is proposed as a way to mitigate the complexity of the scenario.

3.1 Scenario

The scenario evaluated in this thesis is a potential real scenario worked out in collaboration with Kongsberg. The scenario, use cases and problems presented are all elaborations of the challenges faced, and potential solutions.

The network is a mobile tactical network of a number of mobile platforms, communication is key. The platforms are operating in the air, at sea or on the ground, and may be manned or unmanned. The term platform will be used throughout this thesis and will be defined as a system consisting of a communication system, a local system with a specific purpose, e.g. sensors and control mechanisms, and a switch interconnecting all of them.

All platforms are deployed with a specific purpose. It may be to execute an assignment, assist an already ongoing operation, or passively gather data. During the deployment, the platforms require to communicate with a controlling entity and other platforms. The controlling entity will be referred to as C2 and represents the node where the platforms are administered; see C2 in left corner of figure 3.1. This node will also be where the mobile tactical network is connected to the strategic backbone.

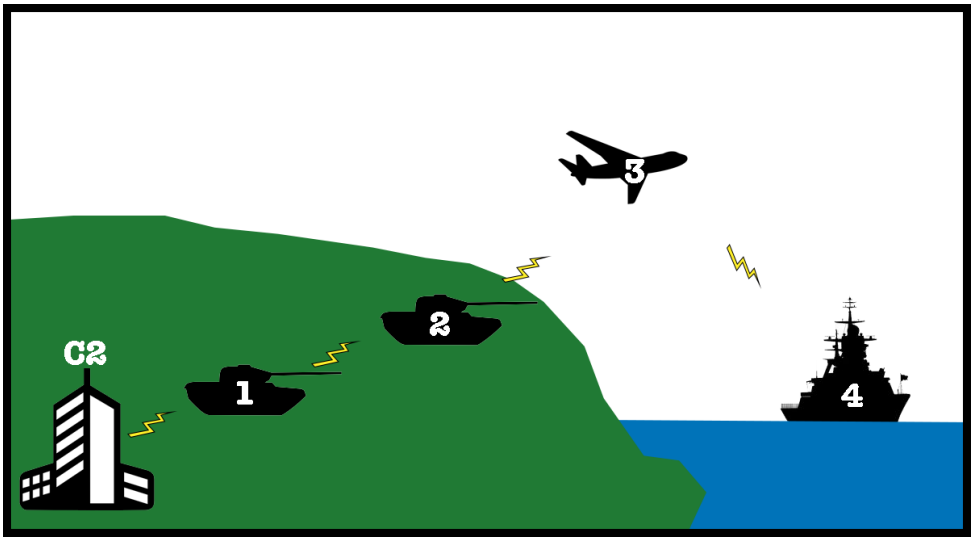

Figure 3.1: Illustration of a mobile tactical network with a variety of nodes. C2 is the commanding entity, node 1 and 2 are ground nodes, node 3 is an aerial node, and node 4 is a surface node.

As the platforms carry out their assignment, they are controlled by, or receive orders from, C2. The multihop nature of the mobile tactical network routes this traffic via the appropriate platforms, to the destination platform. Similarly, data generated by the platforms, will traverse the intermediate platforms to reach the destination. As a consequence, the links close to C2 will experience a heavier load, because of the aggregated data from the outer platforms. Examples of data streams, are video, control data and telemetry data. Certain streams are crucial for the operation of the platform, and other for its current assignment.

The platforms are equipped with military RAT with limited available bandwidth. As the data streams are aggregated throughout the network, some links may be congested. A congested link is critical when several of the platforms have to be controlled from C2, or if a platform is currently executing a mission which requires live video. This can be mitigated by enforcing a certain level of traffic policing, a scheme to differentiate data and police routing of the traffic.

3.2 Heterogeneous Capabilities

To extend the communication capabilities of the platforms, the platforms will be equipped with two RATs. The radios deployed on each of the platforms have differing

and complementing characteristics. They will be referred to as respectively Radio 1 and Radio 2, see figure 3.1. Radio 1 provides superior range, but lacks bandwidth. Radio 2 provides better throughput than Radio 1, but inferior range. An illustration of a platform with two radios are given in figure 3.2, where the local system is connected to both radios.

	Throughput	**Range**	**Frequency**
Radio 1	20 kbps	Long	30-90 MHz
Radio 2	1 Mbps	Med	225-400 MHz

Table 3.1: Comparison of the available communication links and their characteristics. Radio 1 offers low throughput, but long range. Radio 2 provides higher throughput, but shorter range.

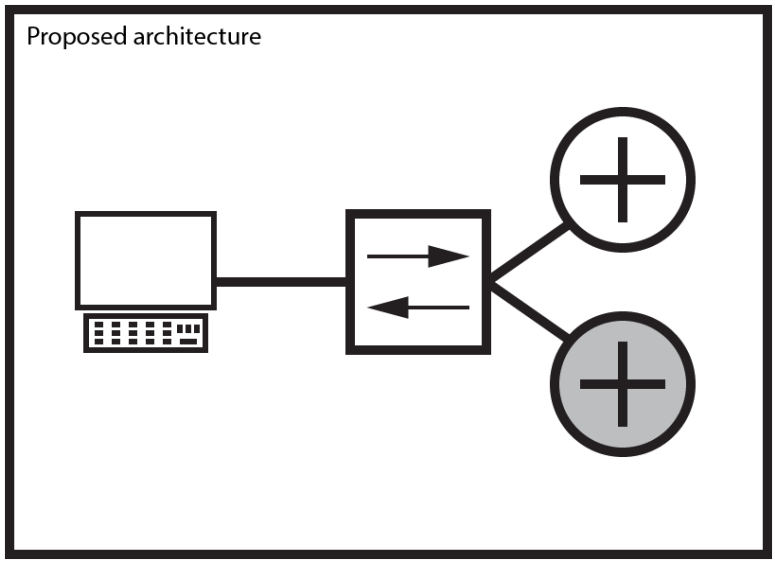

Figure 3.2: The proposed architecture for a platform consisting of 2 RATs, a local system, and a switch interconnecting them.

Multiple RATs provide HetNet capacities. It increases robustness and redundancy in the network. Figure 3.3 illustrates how the platforms in 3.2 can be connected in an example topology. The white radios all have connectivity with each other, while the grey radio are missing one link. Radio 1, with long range, can ensure connectivity, even though the platforms have moved out of range for Radio 2 connectivity. Data can be redundantly transmitted over both channels simultaneously for critical data. Throughput can be increased by utilizing both channels.

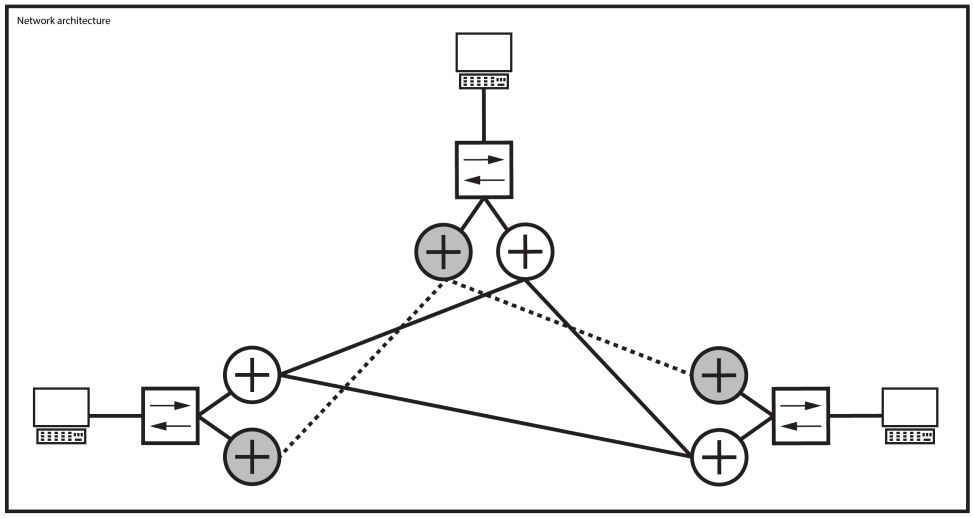

Figure 3.3: Example topology for the platforms illustrated in figure 3.2. Radio 1 (white) are all interconnected, while Radio 2 (grey) only have links from node 1 to 2 and from node 2 to 3.

The radios are not just operating on different channels, they are also running their separate routing software and algorithms. They are running well developed routing protocols for interacting with the network and building topology for the type of network they provide. This means that they are two distinct systems.

Radio 1 is running OSPF on the wired side, and a proprietary routing protocol over the network. Radio 2 is also running OSPF on the wired side, and is currently running Wireless OSPF on the network. Wireless OSPF builds topology by discovering its neighbors and its second hop neighbors from the hello messages sent from its first neighbors.

To enable the platforms to efficiently use the radios, an application or a system that manages the radios has to be present. SDN has the potential to form a such a solution.

3.3 SDN

SDN excel legacy networking with its dynamic nature, and ability to quickly adapt to changes by modifying the policies of the network. It can provide granular control of traffic to shape it according to any available scheme. By implementing the functionality in software, a flexible solution is provided that can easily be extended

with additional services. Building an SDN application can add granular control of the traffic streams traversing the network, and traffic schemes can be worked out for prioritization and filtering. [24] implements a detailed application for traffic engineering. A similar type can be deployed on the platforms.

The core advantages of SDN applies, however, the nature of the mobile tactical network, makes a centralized solution impractical. A centralized solution involves traffic from the SDN Switches to the controller. The control traffic will vary depending the implemented policies and whether the switch is programmed to operate in a reactive or a proactive manner. Nonetheless, the control traffic will represent overhead traffic too significant to sustain. Emitting heartbeat messages, installing new flow rules and handling table misses over TCP is infeasible. The varying quality of the links also poses problems to a centralized solution. The controller can be disconnected as the platform moves in and out of range. The flow entries will then time out, if such a scheme is chosen. These challenges can be mitigated by distributing the intelligence by deploying multiple controllers, as recommended in [24].

Deploying one controller on each platform will ensure constant connectivity between the switch and the controller, but at a cost. The controller will not have an overview of the entire network, and will be ignorant of the existing topology. Without awareness of the current topology, the controller is unable to enforce routing and policing for the network.

The radios have routing protocols specifically designed for their working domain. Years have been spent to tune them for optimal performance. Reconfiguring the network to run SDN instead of the current software, is inpractical. It requires significant effort and time to develop the required controller applications replacing the existing functionality. A hybrid network is a network consisting of both SDN devices and legacy devices, and can be a viable compromise. However, a hybrid network has significant differences from a pure solution. The disadvantages has to be mitigated.

Topology is key to develop a SDN system. Before policies can be enforced, the topology of the network has to be established to know how to configure the traffic flows. Acquiring topology for a network can be constructed in several ways. The whole system does not have to be replaced by SDN switches, but a hybrid solution should be explored. Chapter 4 will evaluate a number of approaches for the physical infrastructure for the platform how it affects its ability to gather topology.

Chapter 4
Architecture design

This chapter proposes three conceptually different models for the architecture of a platform. It walks through the models one by one, and attempts to clarify how they differ in implementation effort, available topology and what grade they can influence routing decisions.

4.1 Proposed models

The physical architecture of the platform greatly influences its ability to develop topology data and understand the network. A controller can only communicate directly with OF enabled devices. This means that if the radios handling routing and transmitting/receiving of traffic do not offer OF support, information about network topology and packets handled, can not be accessed by a regular controller. However it is possible to extend controllers to produce topology in hybrid networks with both SDN and legacy equipment.

This section will present three conceptually different models. Each of the models vary in physical implementation. This affects cost, implementation complexity, and effort required to realize the system. In the platform illustrations (figure 4.1, 4.2, 4.4, 4.3), the Host System represents the local system of the platform. The radios, encapsulated by the dotted lines, represent physical separate systems, currently deployed with a router and a radio interface. The router runs a routing protocol specified for the respective radios. The controller is an SDN controller, and the switch is an SDN switch. The controller and all SDN switches comprises what will be referred to as the SDN system.

4.1.1 Model 1

The first design, figure 4.1, is designed with 1 SDN switch. The radios are left as is, and offer no OF capabilities. This design is simple and cost effective, as it does not require considerable changes to the radios. The SDN switch is connected to both

radios, and can forward packets up or down, that is, to Radio 1 or Radio 2. The SDN system of this model is confined to the controller, and a single SDN switch.

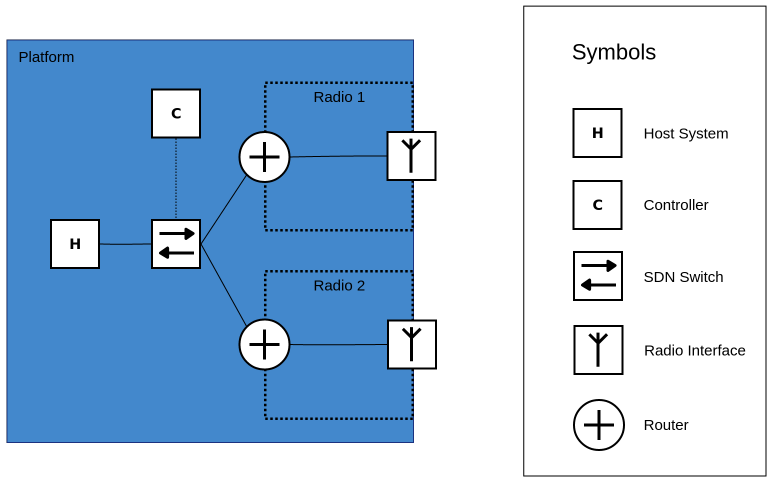

Figure 4.1: Platform design with one SDN switch interconnecting the radios and the local system.

The benefit of this approach is the simple architecture, where the controller will interact with a single switch. The carefully developed and designed radios, do not have to be modified or changed. This is beneficial as the radios are expected to already operate in a resource effective manner, both regarding channel optimization and power management. The main effort of this approach is developing the application to control the SDN switch.

Topology has to be built by the controller to allow it to decide how traffic streams should be handled. Information about the network outside of the platform is not initially available to the controller. The radios are running routing algorithms over the interfaces facing the radio network, but topology messages cannot be intercepted by the SDN system. However, there are two approaches to gather additional topology data for the system. The topology built by the radios, may be accessible and can be utilized by the controller, or the controller can be developed to build its own topology using "hello" beacons.

Gathering the already computed topology from the radios require the radios to have an open API or Command Line Interface (CLI). The controller can then be extended to tap into the database of the radio router and collect topology data. When such an interface is available, an extension to the controller has to be developed

and integrated. Depending on the available interface, this extension may poll the router at a certain interval to update the topology, or listen to events published by the interface. This is determined by the software running in the radio router.

The convergence time for detecting changes in the topology, is determined by the characteristics of the employed routing protocol. Protocols designed for low bandwidth often have higher convergence time than high bandwidth networks. This is because the protocols are designed to minimize the overhead traffic affiliated with building topology, and can do so by sending less updates over the network than in networks operating over higher bandwidth. A difference in expected convergence time for each of the radios should be taken consideration of when implementing the system.

Polling mode adds additional convergence time for detecting changes in the topology, because of the delay between an update in the router database and next poll. This can be mitigated by lowering the polling interval appropriately, but increases internal bandwidth usage and unreasonable high rate may affect energy consumption. The rate of polling should also be considered for each of the radios. Radio 1, with low throughput, will have a lower update rate than Radio 2.

The radios may offer an interface for characteristics for the current links. This is typically given as a quotient, Received Signal Strength Indication (RSSI) value, or Signal-to-noise ratio (SNR). The controller can use this information to rate the links according to a set of specifications.

The second approach is for the controller to build its own topology. To realize this, a protocol has to be developed. A simple hello beacon may suffice in some cases, while for proper robust routing a more complete topology discovery mechanism should be implemented. A hello beacon broadcasted over both radio networks can notify nearby nodes of its existence, its neighbours, and second hop neighbors.

A simple scheme would be to broadcast a hello messages containing the id of the node, and its neighbours. The bit rate for a network consisting **n** nodes, which are sending hello messages to **n - 1** nodes every **t** second, can be calculated like shown in equation 4.1.

The message size will depend on the size of the network, but for this example an average size of 64 bytes will be used.

$$Bitrate = \frac{n * (n-1) * size * 8}{t} \tag{4.1}$$

In a network of 5 nodes and an average size of 64 bytes per packet. The overhead

bit rate is given in equation 4.2. This is not significant for Radio 2, but difficult to handle for Radio 1.

$$Bitrate = \frac{5 * (5-1) * 64 * 8}{15} = 682bit/s \tag{4.2}$$

Running a separate topology algorithm for the SDN system, will also add extra overhead on top of the protocols already running in the radios. Developing different algorithms for the two radios, should be considered to better tune the control traffic to the available bandwidth. The system will also build full topology for both the router and the SDN system. This redundancy can be avoided if the topology is collected like described in the first approach. The trade-of and limitations of reading the topology data from the radios may then be acceptable compared to the additional generated overhead.

The radios, each running their respective routing engines, will process the traffic before it arrives at the SDN switch. If the destination address of the arriving packet does not belong to the intermediary platform, the radio routers, operating in a multihop mode, will forward the traffic to the radio interface destined for its next hop. This means that once traffic originating from the host system of a platform has been sent to the appropriate radio (Radio 1 or 2) by the SDN switch, it is out of control of the SDN system and solely routed by the radios.

This means that when data is generated by a platform, the only way to influence the routing is the initial choice of sending the traffic to Radio 1 or 2. To affect the routing decisions further tunnelling could be used to set up the next destination.

4.1.2 Model 2

The concept for Model 2 is to implement an SDN switch into the radio between the router and the radio interface, see the radios in figure 4.2. This resembles a pipeline structure, where packets move through the router, then the SDN switch, before it is dispatched on the network. The switch is then able to intercept traffic flowing from the radio interface to the radio router, and interact with it. There are several approaches to the infrastructure of this model. Firstly one or both of the radios can have the SDN switch implemented, secondly, the switch interconnecting the system can or cannot be an SDN switch.

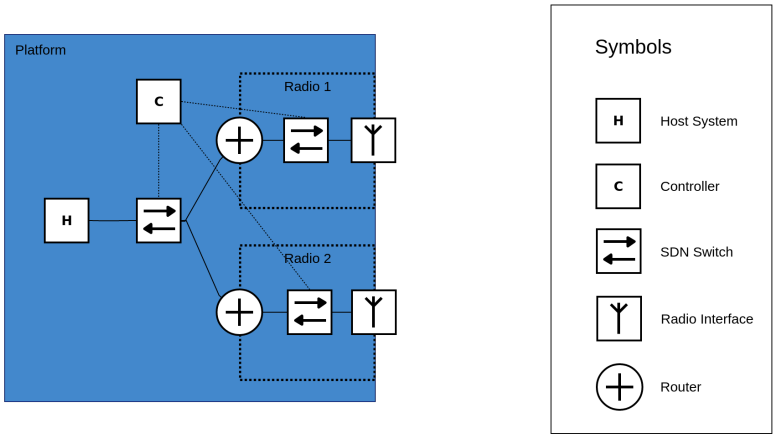

Figure 4.2: Platform design with one SDN switch in each of the radios.

Pipelining the traffic via an SDN switch, enables the controller to manage the traffic flowing between the router and the radio interface in the radio. Packets can be inspected and policies enforced according to any traffic engineering scheme. It can inspect the control traffic generated by the radio router, and either read the packets, forward them, and build its own database, or stop the messages and substitute it with a custom algorithm to build topology.

If the switch is used to tap into the control traffic generated by the router only to read the packets, parts of the routing protocol deployed in the router, has to be implemented in the controller. The current routing protocol, and the capabilities of the controller framework, determine the effort required to implement the controller compatible with the topology messages. SDN controllers support a certain range of protocols, and a frequently deployed protocol is likely to be supported by the chosen controller framework. If the routing protocol is custom made and/or proprietary, the application will have to be extended to support it. Extendability should be emphasized when initially selecting the controller.

Parsing the topology packets, is a similar approach to fetching the topology from the switches in Model 1. It requires an understanding of the protocol to implement the required functionality. However, reading the topology packets in real time, will keep the controller updated in real time. There will be no added propagation delay for topology changes. The controller will now build a redundant database, with the exact same data as the router.

The second option, is to let the SDN switch and the controller run a routing

algorithm itself. Filtering out the control traffic sent from the router, and running complete routing in the controller, requires a considerable implementation effort. This is similar to the second approach for Model 1.

Following the latter option, the router will continuously try to run the routing protocol, but keep being filtered out by the SDN switch. This makes the router redundant. However, the architecture could be a good approach to start developing the radio into supporting SDN and OF. The switch could initially be set to forward all traffic, could then incrementally be augmented to provide new services. This was what OF was initially built for; carving out a piece of the network to be used for development and testing [12].

The center switch for the platform can be an SDN switch or a legacy switch. The platform will have a centralized controller that will have a view over the "whole" network. This is the network of the platform, in addition to whatever topology it has managed to gather. An SDN switch in the center will route the traffic originating from the local system to the appropriate radio according to the policy scheme. A legacy switch would route it according to its local forwarding table. However, the traffic would still be intercepted by the SDN switch in the radios, handled there, and then rerouted to the other radio if necessary.

When the SDN switches are deployed in the radios, the SDN system is able to influence the routing decisions more than in Model 1. After sending the packet to Radio 1 or 2, the routers will attempt to route the packets to a destination according to its current topology. The packets can then be intercepted by the SDN switch before reaching the radio interface. The SDN switch can then alter the packets according to its own topology and traffic engineering requirements, and send it out to the radio interface.

It is also possible to integrate an SDN switch into only one of the radios, illustrated in 4.3. This can be done for example for testing purposes. The SDN system will not have the topology for the radio without the SDN switch. The controller may either let the radio handle the routing without knowledge of its links, or the topology can be collected similar to the first approach in Model 1.

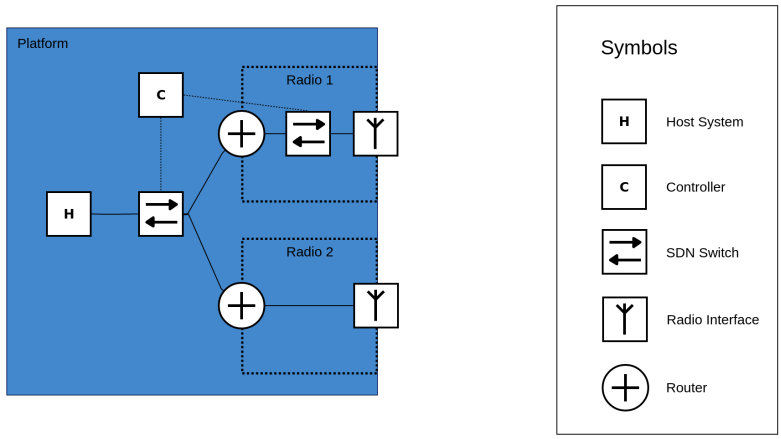

Figure 4.3: Platform design with SDN switch in only one radio.

4.1.3 Model 3

Model 3 represents the most comprehensive and radical change to the existing systems. The routers running in the radios are now substituted by an SDN switch, see figure 4.4. In such a scenario, the controller would be running a full routing suite. Similar to Model 2, where topology was built by the controller, the topology has to be completely built by the SDN system. However, the redundant router running as a zombie, is now omitted. The SDN system is now entirely in control of the platform network, and no work around is needed to route the traffic.

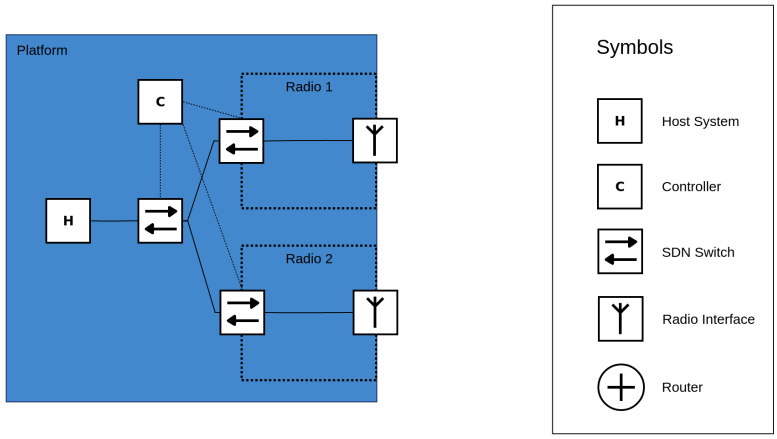

Figure 4.4: Platform design where the routers in the radios are substituted by SDN switches.

Implementing a controller application for what is already present in the radios today will require a significant effort. Replicating the current functionality of the router is possible, and may be beneficial considered the research spent to optimize the protocols.

The controller has to run a controller application for each of the two radios because of the different characteristics for the two radios. As discussed in chapter 3, Radio 2 offers less bandwidth, and cannot be expected to run the same protocol as Radio 1.

Deploying a full SDN solution makes the system more agile and dynamic. A controllers extendability offers graceful ways to develop the system and test out new functionality and protocols.

Chapter 5
Implementation

This chapter will first present a selection of available SDN controllers and the network emulation tool Mininet. Then a simple proof-of-concept for Model 1(figure 4.1), polling topology data, will be presented.

5.0.1 Available controllers

The flourishing scene of SDN has caught the attention of and given birth to several initiatives and foundations. There are a vast range of controllers available, some of which was conceived from the start of OF, while other has just emerged. Controllers differ in implementation language, OF support, north bound API and internal workings. A considerable amount of controllers are open source and is publicly developed as has been the vision of SDN and OF from the start[12]. There are some trends in the field, and some controllers are gaining more momentum than others[10]. Table 5.1 evaluates some of the most prominent open source controllers and their key parameters.

Controller	Language	Active	OpenFlow	Description
Ryu[7]	Python	Yes	1.5	Framework for SDN offering a centralized controller with a python API which is used by applications built on top.
NOX[4]	C++	No	1.0	The first OpenFlow controller. Developed by Nicira Networks and widely used for the initial research of OF. Now discontinued.
POX[4]	Python	Hardly	1.1	A python version of NOX. Eventually gained more support than NOX. Provides a web and python API.
OpenDaylight[9]	Java	Yes	1.4	Alliance founded by Network Equipment Manufacturers (NEMs) offering an SDN platform with a plugin interface for extensions and applications.
OpenContrail[6]	C++/Python	Yes	????	Mainly for cloud networking and Network Function Virtualization (NFV). Offers services both on the control and data plane; OpenContrail Controller and vRouter.
Floodlight[2]	Java	Yes	1.4	Controller with a module system offering Graphical User Interface (GUI), load balancing, and a range of switching apps.
ONOS[5]	Java	Yes	1.5	Open Network Operating System. Hosted by the Linux Foundation to develop a SDN Operating System (OS).
Beacon[1]	Java	No	1.0	Built in Java with OSGi and Spring offering module support. It has been used as the basis for Floodlight[11].

Table 5.1: Comparison of the available open source SDN controllers.

5.1 Ryu

Ryu is an open SDN framework offering a component-based controller. It is supported by Nippon Telegraph and Telephone (NTT) and is currently deployed in the NTT cloud infrastructure. It has also been merged into OpenStack [8], an open source cloud operating system, where it handles network virtualization. The ryu framework is implemented in the control layer, figure 2.2, now illustrated in 5.1 and provides a north-bound API for flexible development of components in the application layer. Ryu is written in python, but offers an API in both python and REST. The REST API is based on JavaScript Object Notation (JSON), which offers the opportunity to implement applications in any given language.

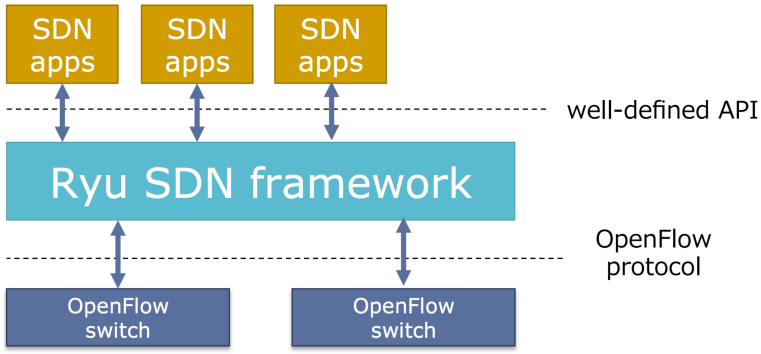

Figure 5.1: Ryu architecture. From [20]. The controller is running in the control layer, and the SDN applications are running in the application layer.

Ryu supports OF versions up to 1.5, along with Nicira Extensions [7]. It also support Netconf, OF-config, Netflow, Simple Network Management Protocol (SNMP) and others. Protocols are defined in separate python files. Currently it support a range of protocols, including IPv4, IPv6, TCP, Border Gateway Protocol (BGP) and Open Shortest Path First (OSPF). The controller can be extended with custom protocols by implementing it, using the existing protocols as examples. Ryu also comes with support for a selection of applications. A switch application with Snort support, offers interoperability with the open source network Intrusion Detection System (IDS) Snort. A BGP speaker for applications supporting BGP routing. And recently it has added a Quagga server, enabling event driven messages from other Quagga daemons.

The rich north-bound interface, the wide support for protocols, packets and applications, along with the simplicity of extending the framework makes Ryu great for rapid development and prototyping. Ryu comes bundled with a range of

applications, including a layer 2 switch, a layer 3 switch, a firewall, an application for graphical representation of the topology, and more. These applications can be used to develop custom and other more complex systems.

Ryu was chosen for the implementation because of the flexibility it provided. The well defined north-bound API, and the ease of developing and prototyping new functionality was valuable. It could easily be integrated with other applications and interfaces.

5.2 Mininet

Mininet [17] is an open source network emulator built in Python. It provides a virtual network which can be used for testing, development and prototyping of new protocols, networks, and SDN controllers. Every component of the network can run on one computer including hosts, switches and controllers. Mininet has been widely used for developing new SDN applications and can be connected to any type of controller over a selected TCP port.

Mininet offers an environment for rapid development and prototyping. A network can be started from the command line with options for customizing the initial network, it can be programmed in Python, or defined using JSON. It comes bundled with several applications for sample setups, and a GUI for designing a network. The Python API is powerful, and is used to configure most aspects of the network, including host configuration, network life cycle, and switch configuration.

A host in Mininet is a linux container with a separate namespace. It acts as a real linux machine with ownership of processes, interfaces, ports, and routing tables. The host can be configured to run custom processes and applications such as routing software or IDSs.

Switches are set up as Linux bridges, or Open Virtual Switch (OVS), open multilayer virtual switches. The Python API can configure these switches and map them to individual controllers, and set the appropriate OF version.

The links connecting the nodes are virtual links over virtual interfaces. The interfaces resides in a network namespace owed by the nodes. To emulate custom characteristics of the links, a Linux tool called Traffic Control (TC) is used. TC is integrated and can be configured by the Python API while programming the networks. Bandwidth, delay, loss, and queue length can easily be altered.

Mininet was chosen for the implementation because of its well defined and documented python API. The API enables simple development of complex networks, which can be changed at run time.

5.3 Implementation

This implementation has been carried out to illustrate the concept explained for the first approach for building topology in Model 1 (4.1). The topology will be built by polling the topology data from a router running a topology algorithm. The goal is not to design a real scenario, but to show an example for how existing topology data can be collected for a platform in the evaluated scenario.

5.3.1 Topology

The network is emulated using Mininet. It consist of three platforms with one router on each platform. The routers represent the router running in a radio and is set up using Quagga[1]. For testing purposes, Quagga is set up to run a simple OSPF set up externally, and with a passive interface internally on the platform, see figure 5.2. A mapping of IPv4 addresses to routers are shown in table 5.2

	External	**Internal**
R1	10.1.100.1/24	10.0.1.1/24
R2	10.1.100.2/24	10.0.2.1/24
R2	10.1.100.3/24	10.0.3.1/24

Table 5.2: Table for IPv4 addresses for the interfaces of the routers in the implemented network.

Internally each platform has 1 host representing the local system of the platform, 1 host running the local controller, and an SDN switch connected in-band to the local controller(see figure 5.2. S4 is set up as a basic layer 2 switch to simulate connection between the platforms. By disconnecting a platform from S4, it can be considered disconnected from the network by the other platforms.

[1]Quagga is a widely deployed open source routing software suite.

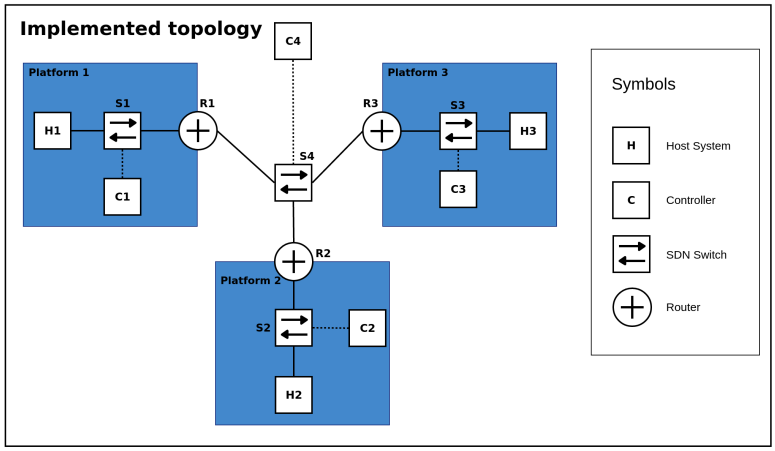

Figure 5.2: Topology of the network implemented using Mininet.

The network programmed in Mininet is given in appendix A. It can be started by running the command in listing 1. The script defines and starts all components of the network, except for the controllers, which will have to be started separately.

Listing 1 Command to start the network.

```
$ sudo python topology_quagga_ospf.py
```

The topology script creates custom classes for the routers (OSPFRouter), switches (OSPFSwitch) and hosts (OSPFHost). The custom classes are created to add additional required functionality to each of the nodes in the network. Listing 2 illustrates how they are instatiated. The *cls* parameter sets the custom class for the nodes.

Listing 2 Instatiation of a switch, a router, and a regular host in the networking script.

```
s1 = self.addSwitch('s1', dpid='00000000000000a1', cls=OSPFSwitch,
↪  protocols='OpenFlow13', inband=True)
r1 = self.addHost(r1name, cls=OSPFRouter,
↪  quaggaConfFile=r1quaggaConf, zebraConfFile=zebraConf,
↪  intfDict=r1intfs)
h1 = self.addHost('h1', cls=OSPFHost, ip='10.0.1.1',
↪  route='10.0.1.254')
```

The router class is set up to configure the host to operate as a Quagga router with the correct settings and start a zebra and OSPF daemon for each of the routers. The commands in listing 3 is executed by the router class. It starts a Zebra and an OSPF daemon with names corresponding to the name of the router.

Listing 3 Command used to configure the routers appropriately.

```
self.cmd( '/usr/lib/quagga/zebra -d -f %s -z %s/zebra%s.api -i
↪  %s/zebra%s.pid' % (self.zebraConfFile, QUAGGA_RUN_DIR, self.name,
↪  QUAGGA_RUN_DIR, self.name) )
self.cmd( '/usr/lib/quagga/ospfd -d -f %s -z %s/zebra%s.api -i
↪  %s/ospfd%s.pid' % (self.quaggaConfFile, QUAGGA_RUN_DIR,
↪  self.name, QUAGGA_RUN_DIR, self.name) )
```

When the network is started, the controller for S4 can be started by running the command in listing 4. S4 is configured to connect to C4 running on port 6656. S4 is a simple layer 2 switch implemented in Ryu. It is set up to print OSPF messages for debugging purposes.

Listing 4 Command to start the controller for S4.

```
$ sudo ryu-manager --ofp-tcp-listen-port 6656 simple_switch.py
```

Next, the local controller for the platforms has to be started. To start the controller in the controller-host on a platform, a shell for the node has to be opened from the Mininet CLI, and then start the Ryu controller in the newly opened shell.

Listing 5 .

```
mininet> xterm h4

$ sudo ryu-manager --ofp-tcp-listen-port 6653 my_switch.py
```

When all the controllers for each of the controller-hosts (C1, C2, and C3), are started. The network is up and running, including hosts, routers, switches, and controllers, on a single machine.

5.3.2 Topology Parser

The controller started in listing 5, is set up to start a layer 2 switch implemented in Ryu. The switch is augmented to spawn a thread running the topology parser.

The data gathered by the topology parser can be utilized as desired to achieve the necessary functionality for the SDN switch.

Listing 6 .

```
self.topology_thread = hub.spawn(self.topology_parser)
```

The topology parser has been implemented using an interface opened by Quagga on port 2604. This interface offers a CLI for acquiring the current data for the router. The topology parser connects to the interface using telnet and executes the appropriate commands. The result is returned as a text string, which is interpreted by the topology parser. The topology parser offers a telnet and a vtysh method to acquire the topology, depending on if the controller is running on the same system as the router or not.

The topology parser has functions to interpret the output of different commands for Quagga. These functions build up a dictionary called *router*(listing 7), which represents the current state of the router. It includes the router id, and information about the attached routers.

Listing 7 Router object of the topology parser from the experiment.

```
{
  "Router ID": "10.1.100.1",
  "Attached Routers": {
    "10.1.100.3": {
      "Connected to": ["10.0.3.0", "10.1.100.1"],
      "Age": "370",
      "Connected": "True"
    },
    "10.1.100.2": {
      "Connected to": ["10.0.2.0", "10.1.100.3"],
      "Age": "371",
      "Connected": "True"
    }
  }
}
```

The topology parser instantiated in the thread spawned by the controller is updated at the polling frequency set by the controller (listing 8). This will ensure the data is updated. The information gathered can be handed to a map for IPs maintained by the router and used as necessary.

Listing 8 .

```
router_parser_handle = topology_parser(self.router_id)
while True:
  router_parser_handle.update()
  print router_parser_handle.router
  time.sleep(15)
```

Chapter 6
Experiences from the Implementation

This chapter will mention what was done in the implementation along with some challenges encountered.

The implementation illustrates that topology data may be collected from a router running a topology discovery algorithm. This is, however, clearly limited by the API offered by the routing software.

The end result was an application that was able to retrieve topology data from the router, structure it for internal use, select relevant information, and serve it to the controller. It illustrates that the approach is a viable option for acquiring topology. However, the implemented application is not a comprehensive solution. To achieve a fully dynamic solution, adequate effort has to be put into development of the application.

The controller has to support gathering the topology data. This could already be implemented and bundled with the controller, or controller has to have support for developing extensions. Ryu turned out to have a recently added zebra server. This was attempted implemented, but lacked documentation and required support at the time. However, Ryu had a well defined API for spawning threads, and creating interfaces between applications.

The routing software has to have an available APIs. A rich and comprehensive API will provide detailed information for the state of the router, but requires a deep understanding of the routing protocol. The data has to be interpreted to collect the data relevant to the controller. The effort for developing a dynamic application to collect this data, increases greatly as the routing protocol increases in complexity. The format for the API is important. The Quagga API used in the implementation, responded with text strings. These text strings had to be structured for further use. Responses in JSON or eXtensible Markup Language (XML) would simplify this process.

Chapter 7
Discussion and Conclusion

This last chapter will first discuss the models in chapter 4. Then state the conclusion for this thesis along with potential future work.

7.1 Discussion

The models proposed in chapter 4 have varying characteristics and advantages. Throughout the models, they present three different ideas for managing topology for the SDN system. The first is to collect already gathered topology data from the radios. This will not generate any additional network traffic, and can be done without modifying the radios. It is explained for Model 1, and Model 2 with only one radio with an SDN switch implemented.

Topology can also built by tapping into the control traffic generated by the routers. This requires the controller to be able to interpret the protocol messages, which is likely for common protocols, but have to be implemented for custom ones. This approach will not generate additional network traffic, but an SDN switch has to be implemented in the radio.

Lastly, topology can be built by developing a custom topology application for the SDN system. This requires much effort, and should be carefully considered before carrying out. The custom topology messages will create additional network traffic if the router is running simultaneously. For both Model 2 and Model 3, the routing protocol for the router can be eliminated.

Routing can be influenced in varying degree. It can either be directly edited by intercepting the traffic and overriding the parameters if necessary, or it can be influenced by utilizing legacy routing schemes to force routing through the network.

The different Models and the appurtenant approaches to building topology, requires varying implementation effort. Collecting the already existing topology

depends on the API offered by the routing software, but it has to be parsed and interpreted before it is integrated into the topology application. The application interpreting the data, will have to be developed unless the controller framework offers an interface to the router. Listening to the topology messages of the router, requires the controller to understand the protocol, which may also be considered as a major task. However, developing an application to create topology will requires the most effort.

Model 1, which doesn't require any modifications to the radios, would be the natural first step when moving towards a full SDN system. It provides some basic functionality, while not requiring changes to existing equipment. The next step would then be to implement an SDN switch into the radio. Initially, implementing it will not affect the system at all. The SDN system can then be extended with new functionality, and can be used for testing, Eventually, the SDN switch can be implemented to handle most of the functionality of the radio. Lastly the transition can be smooth to a full SDN system with the same capabilities as the legacy network, but with the added properties of SDN.

7.2 Conclusion

In this study the scenario encountered by Kongsberg, involving heterogeneous mobile nodes, has been elaborated, and the characteristics of the network has been structured. It was encountered that the first step for transitioning towards an SDN network, was to build topology for a SDN controller. Then three conceptually different architecture models are presented. The architecture of the models influences its ability to build the topology for the SDN application. It also affects implementation effort and the influence over the routing decisions. A proof of concept was developed for an approach that collects topology data from a router. It turned out to be a feasible solution to collect this information, but required extensive knowledge of the deployed routing protocol. Lastly it is argued that the models can be used to incrementally implement SDN capabilities into the evaluated platform.

7.3 Future Work

Developing a more dynamic SDN application to parse the topology data from the routing software would make the application more applicable to a changing environment. Developing and evaluating how SDN applications can use the gathered topology, would be the next step towards a software defined heterogeneous mobile tactical network.

References

[1] Beacon. https://openflow.stanford.edu/display/Beacon/Home. Accessed: 2017-06-06.

[2] Floodlight openflow controller - project floodlight. http://www.projectfloodlight.org/floodlight/. Accessed: 2017-06-06.

[3] Mobile ad-hoc networks (manet): About. https://datatracker.ietf.org/wg/manet/about/. Accessed: 2017-05-23.

[4] The nox/pox controller. https://github.com/noxrepo/nox https://github.com/noxrepo/pox. Accessed: 2017-06-06.

[5] Onos - a new carrier-grade sdn network operating system designed for high availability, performance, scaleout. http://onosproject.org/. Accessed: 2017-06-06.

[6] Opencontrail. http://www.opencontrail.org/. Accessed: 2017-06-06.

[7] The opendaylight platform. https://www.opendaylight.org/. Accessed: 2017-06-06.

[8] Openstack. https://www.openstack.org/. Accessed: 2017-06-06.

[9] Ryu sdn framework. https://osrg.github.io/ryu/. Accessed: 2017-06-06.

[10] Cohn, M. (2016). 2016 SDN Controller Landscape. Is there a Winner? ONF Tutorial Day at ONS, Accessed: 2017-06-06.

[11] Erickson, D. (2013). The Beacon OpenFlow Controller. In *HotSDN*. ACM.

[12] Goransson, P. and C. Black (2014). *Software Defined Networks: A Comprehensive Approach* (1st ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Accessed: 2017-06-06.

[13] Helou, M. E., S. Lahoud, M. Ibrahim, and K. Khawam (2013, April). A hybrid approach for radio access technology selection in heterogeneous wireless networks. In *European Wireless 2013; 19th European Wireless Conference*, pp. 1–6. Accessed: 2017-05-24.

[14] Hossain, E. (2008). *Heterogeneous Wireless Access Networks: Architectures and Protocols*. Springer Science & Business Media. Accessed: 2017-04-20.

[15] Khawam, K., M. Ibrahim, J. Cohen, S. Lahoud, and S. Tohme (2011, June). Individual vs. global radio resource management in a hybrid broadband network. In *2011 IEEE International Conference on Communications (ICC)*, pp. 1–6. Accessed: 2017-05-24.

[16] Kure, Ø. and I. Sorteberg (2004). Network architecture for network centric warfare operations. Technical report, Norwegian Defence Research Establishment.

[17] Lantz, B., B. Heller, and N. McKeown (2010). A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, pp. 19:1–19:6. ACM. Accessed: 2017-06-20.

[18] Macker, J. P. and D. S. M. Corson (1999, January). Mobile ad hoc networking (manet): Routing protocol performance issues and evaluation considerations. RFC 2501. Accessed 2017-05-22.

[19] McKeown, N., T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner (2008, March). Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*(2), 69–74. Accessed: 2017-03-10.

[20] Ohmura, K. (2013). OpenStack/Quantum SDN-based network virtulization with Ryu. Accessed: 2017-06-20.

[21] O.N.F. (2012). Software-defined networking: The new norm for networks. *ONF White Paper*. Accessed: 2017-06-07.

[22] Open Networking Foundation (2015). OpenFlow Switch Specification 1.5.1. Technical report. Accessed: 2017-06-12.

[23] SDxCentral (2016). Special Report : OpenFlow and SDN – State of the Union. Technical report, SDNCentral LLC. Accessed: 2017-06-08.

[24] Skappel, H. F. (2016, June). Traffic policing in dynamic military networks using software defined networking. Master's thesis, Norwegian University of Science and Technology. Accessed: 2017-05-10.

Appendix A

Mininet Topology Script

```
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.cli import CLI
from mininet.log import setLogLevel, info, debug
from mininet.node import Host, RemoteController
from mininet.node import Controller, OVSController
from mininet.node import OVSKernelSwitch

QUAGGA_DIR = '/usr/lib/quagga'
# Must exist and be owned by quagga user (quagga:quagga by default on
  ↪ Ubuntu)
QUAGGA_RUN_DIR = '/var/run/quagga'
# To avoid permission denied problems
# Folder bust be owned by quagga:quaggavty
# Files must be owned by quagga:quagga
# Files must be executable
CONFIG_DIR = '/etc/quagga/configs'

class InbandController( RemoteController ):
  def __init__(self, *args, **kwargs):
    RemoteController.__init__(self, *args, **kwargs)

  def checkListening( self ):
    "Overridden to do nothing."
    return

class OSPFHost(Host):
  def __init__(self, name, ip, route, *args, **kwargs):
    Host.__init__(self, name, ip=ip, *args, **kwargs)
```

```
29
30      self.route = route
31
32    def config(self, **kwargs):
33      Host.config(self, **kwargs)
34
35      debug("configuring route %s" % self.route)
36      self.cmd('ip route add default via %s' % self.route)
37
38  class OSPFSwitch(OVSKernelSwitch):
39    def __init__(self, name, *args, **kwargs):
40      OVSKernelSwitch.__init__(self, name, *args, **kwargs)
41
42    def start(self, a):
43      return OVSKernelSwitch.start(self, [cmap[self.name]])
44
45  class OSPFRouter(Host):
46    def __init__(self, name, quaggaConfFile, zebraConfFile, intfDict,
    ↪  *args, **kwargs):
47      Host.__init__(self, name, *args, **kwargs)
48
49      self.quaggaConfFile = quaggaConfFile
50      self.zebraConfFile = zebraConfFile
51      self.intfDict = intfDict
52
53    def config(self, **kwargs):
54      Host.config(self, **kwargs)
55      self.cmd('sysctl net.ipv4.ip_forward=1')
56
57      for intf, attrs in self.intfDict.items():
58        self.cmd('ip addr flush dev %s' % intf)
59        if 'mac' in attrs:
60          self.cmd('ip link set %s down' % intf)
61          self.cmd('ip link set %s address %s' % (intf, attrs['mac']))
62          self.cmd('ip link set %s up' % intf)
63        for addr in attrs['ipAddrs']:
64          self.cmd('ip addr add %s dev %s' % (addr, intf))
65
66      self.cmd('/usr/lib/quagga/ospfd -d -f %s -z %s/zebra%s.api -i
    ↪  %s/ospfd%s.pid' % (self.quaggaConfFile, QUAGGA_RUN_DIR,
    ↪  self.name, QUAGGA_RUN_DIR, self.name))
```

```
67
68   def terminate(self):
69     self.cmd("ps ax | egrep 'ospfd%s.pid|zebra%s.pid' | awk
   ↪ '{print$1}' | xargs kill" % (self.name, self.name))
70     Host.terminate(self)
71
72 class OSPFTopo( Topo ):
73
74   def build( self ):
75     s1 = self.addSwitch('s1', dpid='00000000000000a1',
   ↪ cls=OSPFSwitch, protocols='OpenFlow13', inband=True)
76     s2 = self.addSwitch('s2', dpid='00000000000000a2',
   ↪ cls=OSPFSwitch, protocols='OpenFlow13', inband=True)
77     s3 = self.addSwitch('s3', dpid='00000000000000a3',
   ↪ cls=OSPFSwitch, protocols='OpenFlow13', inband=True)
78     s4 = self.addSwitch('s4', dpid='00000000000000a4',
   ↪ cls=OSPFSwitch, protocols='OpenFlow13')
79
80     zebraConf = '%s/zebra.conf' % CONFIG_DIR
81
82     # Router 1
83     r1name = 'r1'
84     r1eth0 = { 'mac' : '00:00:00:00:01:01',
85              'ipAddrs' : ['10.0.1.254/24'] }
86     r1eth1 = { 'mac' : '00:00:00:00:01:02',
87              'ipAddrs' : ['10.1.100.1/24'] }
88     r1intfs = { 'r1-eth0' : r1eth0,
89               'r1-eth1' : r1eth1}
90     r1quaggaConf = '%s/quagga1.conf' % (CONFIG_DIR)
91     r1 = self.addHost(r1name, cls=OSPFRouter,
   ↪ quaggaConfFile=r1quaggaConf, zebraConfFile=zebraConf,
   ↪ intfDict=r1intfs)
92
93     # Router 2
94     r2name = 'r2'
95     r2eth0 = { 'mac' : '00:00:00:00:02:01', 'ipAddrs' :
   ↪ ['10.0.2.254/24'] }
96     r2eth1 = { 'mac' : '00:00:00:00:02:02', 'ipAddrs' :
   ↪ ['10.1.100.2/24'] }
97     r2intfs = { 'r2-eth0' : r2eth0, 'r2-eth1' : r2eth1}
98     r2quaggaConf = '%s/quagga2.conf' % (CONFIG_DIR)
```

```
99      r2 = self.addHost(r2name, cls=OSPFRouter,
    ↪ quaggaConfFile=r2quaggaConf, zebraConfFile=zebraConf,
    ↪ intfDict=r2intfs)
100
101     # Router 3
102     r3name = 'r3'
103     r3eth0 = { 'mac' : '00:00:00:00:03:01', 'ipAddrs' :
    ↪ ['10.0.3.254/24'] }
104     r3eth1 = { 'mac' : '00:00:00:00:03:02', 'ipAddrs' :
    ↪ ['10.1.100.3/24'] }
105     r3intfs = { 'r3-eth0' : r3eth0, 'r3-eth1' : r3eth1}
106     r3quaggaConf = '%s/quagga3.conf' % (CONFIG_DIR)
107     r3 = self.addHost(r3name, cls=OSPFRouter,
    ↪ quaggaConfFile=r3quaggaConf, zebraConfFile=zebraConf,
    ↪ intfDict=r3intfs)
108
109     # Add links to their separate networks
110     self.addLink(r1, s1)
111     self.addLink(r2, s2)
112     self.addLink(r3, s3)
113
114     # Add links to connect the OSPF network
115     self.addLink(r1, s4)
116     self.addLink(r2, s4)
117     self.addLink(r3, s4)
118
119     # Add hosts to each platform
120     h1 = self.addHost('h1', cls=OSPFHost, ip='10.0.1.1',
    ↪ route='10.0.1.254')
121     h2 = self.addHost('h2', cls=OSPFHost, ip='10.0.2.1',
    ↪ route='10.0.2.254')
122     h3 = self.addHost('h3', cls=OSPFHost, ip='10.0.3.1',
    ↪ route='10.0.3.254')
123     self.addLink(h1, s1)
124     self.addLink(h2, s2)
125     self.addLink(h3, s3)
126
127     # Add hosts for controllers for each platform
128     h4 = self.addHost('h4', ip='10.0.1.2')
129     h5 = self.addHost('h5', ip='10.0.2.2')
130     h6 = self.addHost('h6', ip='10.0.3.2')
```

```
131     self.addLink(h4, s1)
132     self.addLink(h5, s2)
133     self.addLink(h6, s3)
134 
135 topos = { 'ospf' : OSPFTopo }
136 
137 # One controller for taking care of the OSPF network and
138 # one for the switches outside of the OSPF network.
139 # c1 running inside the OSPF network. Runs a simple switch.
140 c1 = InbandController(name='c1', ip='10.0.1.2', port=6653)
141 c2 = InbandController(name='c2', ip='10.0.2.2', port=6654)
142 c3 = InbandController(name='c3', ip='10.0.3.2', port=6655)
143 c4 = InbandController(name='c4', ip='127.0.0.1', port=6656)
144 # ryu-manager --ofp-tcp-listen-port 6656 simple_switch.py
145 
146 cmap = {'s1': c1, 's2': c2, 's3': c3, 's4': c4}
147 
148 if __name__ == '__main__':
149   setLogLevel('debug')
150   topo = OSPFTopo()
151 
152   net = Mininet(topo=topo, controller=None)
153 
154   for controller in cmap:
155     net.addController(cmap[controller])
156 
157   net.start()
158   # Set IP to be able to communicate with the in-band controller
159   net.getNodeByName('s1').cmd('ifconfig s1 inet 10.0.1.10')
160   net.getNodeByName('s2').cmd('ifconfig s2 inet 10.0.2.10')
161   net.getNodeByName('s3').cmd('ifconfig s3 inet 10.0.3.10')
162   net.getNodeByName('s4').cmd('ifconfig s4 inet 10.1.100.4')
163 
164   CLI(net)
165 
166   net.stop()
167 
168   info("done\n")
```