NTNU
Norwegian University of
Science and Technology

Evaluation of SDN in Small Wireless-capable and Resource-constrained Devices

Million Aregawi Beyene

Master of Telematics - Communication Networks and Networked Services
Submission date: June 2017
Supervisor: Yuming Jiang, IIK
Co-supervisor: David Palma, IIK

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

NTNU – Trondheim
Norwegian University of
Science and Technology

Evaluation of SDN in Small Wireless-capable and Resource-constrained Devices

Million Aregawi Beyene

Submission date: June 26, 2017
Responsible professor: Yuming Jiang, IIK
Supervisor: David F Palma, IIK

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

Title: Evaluation of SDN in Small Wireless-capable and Resource-constrained Devices

Student: Million Aregawi Beyene

Problem description:

Accompanied by the developments in miniaturization and wireless technologies, the number of devices becoming part of the global network is growing rapidly and we are heading into the era of Internet of Things (IoT). Considering their mobile, dynamic and resource-constrained nature, the network management of billions of these tiny connected devices will obviously be very challenging. Researchers in the field are recently suggesting new paradigms that would make networks more flexible, programmable, easy to manage and open from vendor lock-in. Software Defined Networking (SDN) is one of the solutions that has got much attention from researchers. SDN decouples the network into data plane and control plane. The programmable control plane has all the intelligence about the whole network and decides how packets should be routed. The data plane is only responsible for packet forwarding using the rules installed by the control plane. Much of the research about SDN has been focused on its use in infrastructure-based and wired networks. However, with the emergence of IoT and related technologies, the research community has started proposing SDN solutions for a network of small wireless-capable and resource-constrained devices. Software Defined Networking for WIreless SEnsor Networks (SDN-WISE) is one of the proposed solutions. Since the nodes in IoT network have limited battery life, memory, and processing capacity, the performance and efficient utilization of the limited resources has to be optimized. The main objective of this master's thesis is to deploy SDN (mainly SDN-WISE) in IoT or similar networks and carry out performance evaluation. The plan is to set up a prototype and experimental testbed consisting of a network of sensor nodes using real hardware or in a simulator environment. After the testbed is set up successfully, then performance analysis of the prototype will be carried out considering different scenarios.

Responsible professor: Yuming Jiang, IIK

Supervisor: David F Palma, IIK

Abstract

Software Defined Networking (SDN) has recently emerged as the promising network architecture paradigm that is believed to revolutionize the networking world by making it more intelligent, flexible and programmable. SDN has mainly been researched for wired and data center networks. However, aligned with the evolution of Wireless Sensor Networks (WSN), Internet of Things (IoT) and similar technologies, we believe that the principles and potentials of SDN can be applied to a network of resource-constrained devices. The main objective of this thesis is to test if SDN can be deployed in a network of small wireless-capable and resource-constrained devices by setting up an experimental testbed, and then to evaluate its performance. The methodology used is real hardware-based experimentation and measurement instead of simulation. The rationale behind the choice is so as to undertake the evaluation in an environment that depicts reality and get reasonable results. A literature review of previous studies shows that most of the research works done are at an architectural or prototype level. However, the authors who suggested SDN for WIreless SEnsor Network (SDN-WISE) tried to implement their solution and made their source code publicly available. This thesis work uses SDN-WISE codebase as a starting point and further implementation of missing components and improvement of existing functionalities is carried out. A working hardware-based WSN testbed that uses SDN paradigm is implemented as one of the contributions of this thesis work. Software related contributions include implementation of a dynamic topology discovery, set of serial commands to program sensor nodes, a Python-based SDN controller, and a simple serial line based communication protocol between WSN and the SDN controller residing in a general purpose computer. The performance of the developed testbed is evaluated under different scenarios. The effects of topology, hop distance, and rate of packet arrival on the performance of the testbed have been analyzed. As the topology grows with more sensor nodes and becomes more complex, the performance declines. The same is true when the hop distance and rate of packet arrivals increase also. The ease of management, flexibility, and programmability provided by SDN, as witnessed in the experiments, convinced us to recommend and propose SDN as a viable solution for a network of small wireless-capable and resource-constrained devices.

Preface

This thesis report is submitted in partial fulfillment to the completion of my MSc study in Telematics - Communication Networks and Networked Services at the Department of Information Security and Communication Technology, Norwegian University of Science and Technology (NTNU). Postdoc David Palma has been supervising the work and Professor Yuming has been the responsible professor. The work was carried out from Jan 23 - Jun 26, for 22 weeks in the spring semester of 2017. To the best of my knowledge, this report is original product of my thesis work except where acknowledgements and references are made.

Million Aregawi Beyene

June 2017
Trondheim, Norway

Acknowledgement

I would like to thank my supervisor David Palma and responsible professor Yuming Jiang for their continuous support and guidance. I am also much indebted to the Norwegian government and people for funding my master's study. Staff members of our department Laurent Paquereau, Pål Sturla Sæther, Mona Nordaune and Randi Fløne have also been very helpful in facilitating administrative support, providing working space and supplying required materials. My friends have also been encouraging me to work harder and constantly reminding me that I can do it. I am thankful for all their support. Last but not least, I am really grateful to my family for always being the source of my inspiration throughout my life, and for always being on my side.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

API Application Programming Interface.

CAPEX Capital Expenditure.

CPU Central Processing Unit.

FWD Forwarding Layer.

IEEE Institute of Electrical and Electronics Engineers.

INPP In-Network Packet Processing.

IoT Internet of Things.

IP Internet Protocol.

IPv4 Internet Protocol version 4.

IPv6 Internet Protocol version 6.

LR-WPAN Low Rate - Wireless Personal Area Network.

M2M Machine-to-Machine.

MAC Media Access Control.

MCU Micro Controller Unit.

NTNU Norwegian University of Science and Technology.

NXH Next Hop to Sink.

ONF Open Networking Foundation.

OPEX Operational Expenditure.

OS Operating System.

PHY Physical Layer.

PY-SDN-WISE Python-based SDN-WISE Controller.

RAM Random Access Memory.

RPL Routing Protocol for Low-power and Lossy Networks.

RSSI Received Signal Strength Indicator.

SDCSN Software Defined Clustered Sensor Network.

SDN Software Defined Networking.

SDN-WISE Software Defined Networking - WIreless SEnsor Network.

SDWN Software Defined Wireless Network.

SOF Sensor OpenFlow.

TCP Transmission Control Protocol.

TCP/IP Transmission Control Protocol/Internet Protocol.

TD Topology Discovery.

UART Universal Asynchronous Receiver/Transmitter.

USB Universal Serial Bus.

WMN Wireless Mesh Network.

WPAN Wireless Personal Area Network.

WSN Wireless Sensor Network.

Chapter 1
Introduction

One of the exciting and, at the same time, challenging thing about technology is that it has to keep evolving to address emerging problems and become a tool for a better world. Accompanied by advancement in miniaturization and wireless technologies, almost everything is becoming part of the global network as we witness technologies like Internet of Things (IoT), Machine-to-Machine (M2M) and Wireless Sensor Networks (WSN) [FS17]. Since these networks are made up of elements that have limited battery life and processing power, there is a need for better utilization of their resources. Moreover, because of their large number and mobile nature, they demand a very flexible, intelligent, robust and effective network management. Compared to other areas of computing, the networking technology has, for decades, not shown an impressive progress. This is mainly attributed to the proprietary nature of the hardware and software used in the platform. The networking industry is dominated by some giant vendors that share a large portion of the market. With the proprietary nature of most networking devices and lack of open standards, there are only limited interfaces to the internal workings of the devices provided by vendors, and creating a vendor lock-in of networks [SSP16]. This hinders the evolution of networks since solution makers do not get the freedom and handy tools to develop and implement their solutions. If a developer wants to implement some kind of functionality or solution in their networks, they can only use the interfaces or set of commands that are provided by the vendor. It is to solve this lock-in and sluggish evolution in the networking world that the SDN paradigm has been introduced. SDN has been a revolutionizing concept in computer communications, and it is a promising paradigm that is believed to simplify management of networks and minimize vendor lock-in by adopting open standards and specifications [KRV+14]. In spite of that, SDN has mainly been researched for its use in data-centers and infrastructure-based networks [AMN+14]. Protocols and standards, like that of OpenFlow [MAB+08], have been defined for SDN in wired networks. However, SDN can also be used to simplify network configuration and resource management in resource-constrained networks [CGMP12]. Nowadays, SDN is being considered and proposed for small

wireless-capable and resource-constrained devices such as sensor nodes. One of the recent works to this effort is SDN-WISE [GMMP15c]. The SDN-WISE authors have even tried to develop a codebase that enables SDN to be deployed in WSN. Aligned with the evolution of IoT, WSN and similar technologies, this thesis focuses on the evaluation of SDN in small wireless-capable and resource-constrained devices. So the main objective of this thesis is to investigate if SDN can be deployed in a real sensor network, and then set up an experimental testbed and evaluate its performance. Researching SDN solutions for a network of resource-constrained devices is very important as more devices are becoming part of the global network and the need for very intelligent, robust, programmable and manageable architectures that can satisfy the demands of these future networks is also growing. Intelligent routing applications that take into account the battery levels of sensor nodes can be implemented to potentially increase the lifetime of the networks [BK16]. We hope this thesis work will open the door for further research in exploiting the potentials of SDN to solve some of the inherent problems in resource-constrained networks by providing better management, robustness, and programmability.

1.1 Motivation

Some of the motivating factors for carrying out this thesis work are:

- SDN is one of the recently introduced common buzzwords in the network world. It is branded to have many good features that can transform existing network paradigms. So we wanted to explore and study more about SDN.
- Technologies like IoT, WSN, M2M are growing in scope. They are used in different application scenarios including in agriculture, military, health-care and energy sectors [BMRO16]. Moreover, resource-constrained devices are the building components of IoT, WSN, M2M networks. These devices demand a very robust and intelligent management that can take their limited resources into account. SDN can be a possible solution.
- Most of the research about SDN focuses on its use in wired and infrastructure-based networks. Therefore, we are curious to investigate how SDN can be deployed in networks of small wireless-capable and resource-constrained devices.
- Our inclination for applied and practical research is also another driving factor. This thesis work has both software implementation and experimentation, which makes the research work interesting.

1.2 Objectives

The main objectives of this thesis work are:

- To set up a real hardware-based experimental testbed of WSN that works according to the principles of SDN paradigm.
- To evaluate the performance of the testbed by considering different scenarios.

Specific objectives that have to be met in order to achieve the main objectives:

- To review existing SDN solution for a network of small wireless-capable and resource-constrained devices, and choose one that can be used in our testbed.
- To list out the limitations of chosen SDN solution.
- To implement the missing functionalities and components that would make the testbed complete.

1.3 Problem Statement

This thesis will try to answer the following questions:

- Are there SDN solutions for a network of resource-constrained devices?
- If so, can we set up a real hardware-based experimental testbed that implements SDN in a network of resource-constrained devices?
- How is the performance of the implemented testbed?
- How is the performance affected by the topology of the network?
- What benefits can we get from SDN being deployed in a network of small wireless capable and resource-constrained devices?

1.4 Thesis Contribution

We have done both implementation of a testbed and evaluation of its performance in this thesis work. The details of the achieved contributions will be discussed later in different sections of this thesis report. The main contributions can be summarized as follows:

- Set up of a full stack hardware-based testbed of WSN that works according to the principles of SDN. We developed our own Python-based SDN controller to make the testbed complete.

- Extending the codebase of SDN-WISE to use dynamic neighbor and topology discovery which is essential in case of mobility that is inherent to sensor networks.
- Defined new message formats and a simple custom protocol that helps the SDN controller and sink node to communicate.
- A set of commands are also defined that help to configure the sensor nodes and communicate with them using a serial communication.
- Finally, performance analysis is carried out in the custom testbed and the results under different scenarios are analyzed.

The software implementation contributions we made are attached in Appendix A. The whole testbed implementation can also be accessed from Github [1].

1.5 Thesis Structure

This thesis report has 7 chapters in total, including this introductory chapter. The remaining part of the report is organized as follows:

- **Chapter 2: Background and Literature Review** - gives general overview and definition to some of the technologies that are a basis to this thesis work. A literature review of related works is also presented.
- **Chapter 3: Methodology and Testbed Implementation** - discusses the research methodologies used, followed by a detailed description of how the testbed is implemented. The main implementation contributions are presented in this chapter.
- **Chapter 4: Experiment Setup** - this chapter discusses the experimental setup, the different metrics and scenarios used for our evaluation.
- **Chapter 5: Results** - The results obtained from the experimental measurements are presented.
- **Chapter 6: Discussion** - a discussion and analysis of the results obtained will be discussed in this chapter.
- **Chapter 7: Conclusion** - this chapter will include concluding remarks from the thesis work and some recommendations.

[1] https://github.com/miarcompanies/sdn-wise-contiki

Chapter 2
Background and Literature Review

This chapter is intended to give the reader a general understanding of some of the technologies and terms that are used in this thesis work. A literature review of related works is also discussed.

2.1 Resource-constrained Devices

Resource-constrained devices are small devices that have limited processing capability (CPU and clock rate), storage (RAM and Flash memory), and they often run on batteries with limited lifetime [20114]. These devices can connect to each other and form networks that are used in different application areas. They are the building blocks of emerging network technologies like WSN, IoT, and M2M networks.

2.2 Software Defined Networking (SDN)

Networks have two main planes of operation: Control plane and Data plane. The control plane is responsible to make decisions on how packets should be forwarded, and the data plane actually forwards packets based on the decisions (rules) made by the control plane [KRV+14]. In traditional networks, both the control plane and data plane are coupled within the same hardware. On the other hand, SDN is a new network architecture paradigm where the control plane and data plane of the traditional network are physically separated, and the overall decisions of the network are made by a centralized controller. Open Networking Foundation (ONF)[Opea] is a user-driven organization dedicated to the promotion and adoption of SDN through open standards development. According to ONF[Opeb], the formal definition of SDN is *"... an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. The architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services."* SDN provides

an open platform where different network functionality can easily be implemented and put into operation. The logical view of SDN architecture and its layers (planes) is shown in Figure 2.1. Through the centralized controller and its APIs, SDN provides

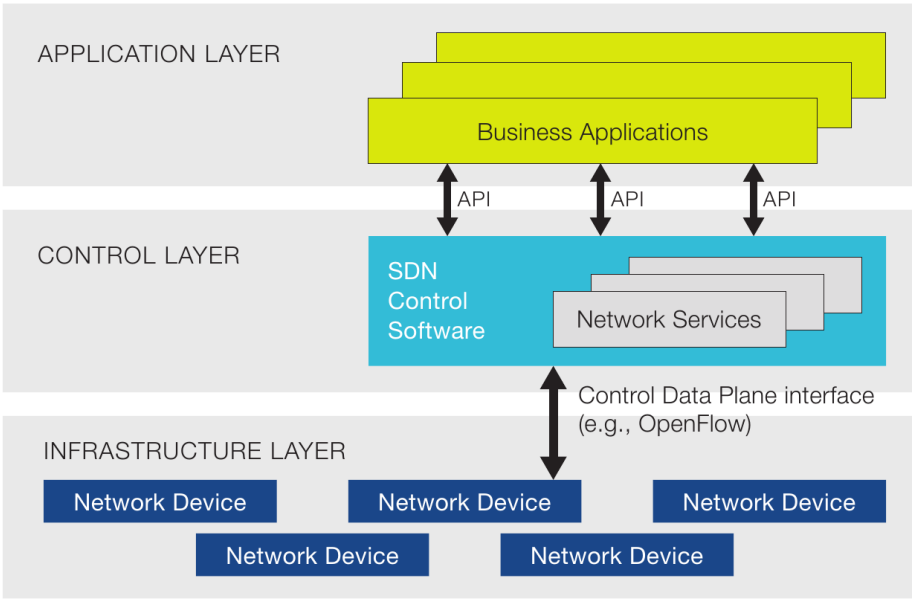


Figure 2.1: SDN Architecture [O.N12]

a single point of control where flexible, robust and scalable applications can be developed, following the best practices of application development, to manage the devices in the network. This simplifies the management and configuration of the network. OpenFlow is the defacto protocol that is used to communicate the control plane and data plane of the SDN architecture. OpenFlow allows the applications in the controller to directly access and manipulate the forwarding rules of the data plane devices. The data plane devices forward packets based on flow rules. The flow rules are stored in a tabular structure on the forwarding devices. Each flow rule has match criteria and set of associated actions. All packets that match the flow criteria are forwarded according to the corresponding action in the flow entry. Matching can be based on IP source, IP destination, MAC source, MAC destination, TCP destination port, TCP source port and other fields. Some of the possible actions defined by OpenFlow include dropping a packet, broadcasting a packet or forwarding it to any of the forwarding device's ports. Table 2.1 and Figure 2.2 show a comparison of traditional and SDN network architectures.

Traditional Networks	SDN
Control plane and data plane coupled within same hardware	Control plane and data plane decoupled to separate hardwares
Distributed intelligence	Centralized intelligence and network view
Vendor lock-in	Open standards E.g: OpenvSwitch and OpenFlow
Configuration and policy implementation is tedious and difficult	Programmable and flexible
Higher CAPEX and OPEX	Lower CAPEX and OPEX

Table 2.1: Comparison of Traditional and SDN Network Paradigms

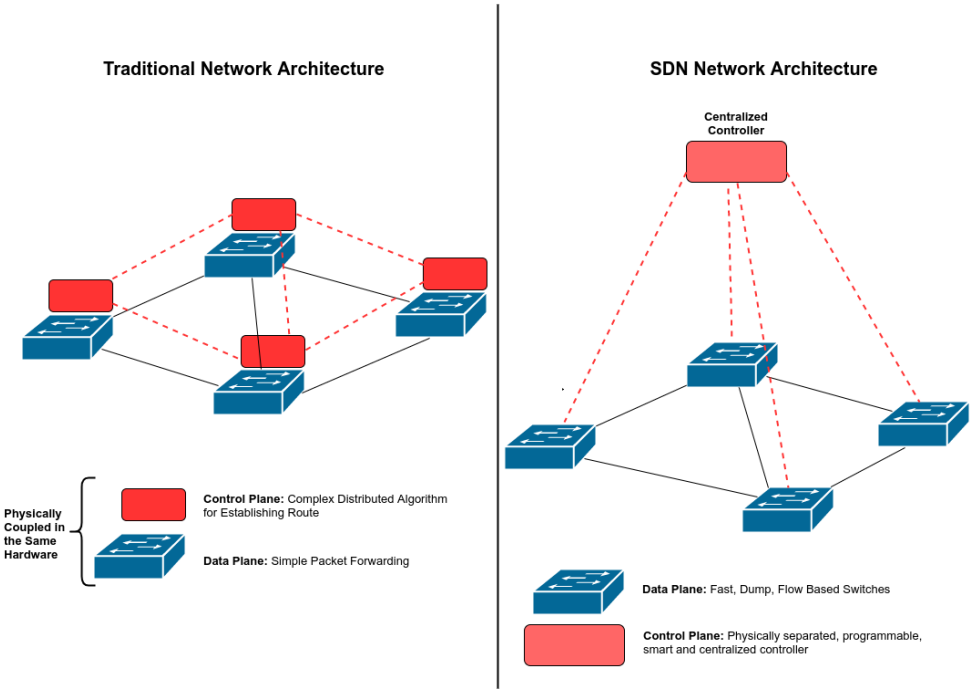


Figure 2.2: Comparison of Traditional vs SDN Network Architectures

2.3 Wireless Sensor Networks (WSN)

WSN is a network of sensor nodes that are able to collect information from their surrounding such as temperature, pressure, sound, light, motions and so on [DGAM14]. A typical WSN has the architecture shown in Figure 2.3. Each sensor node has a processing unit, sensors/actuators to interact with the environment, and a wireless interface that is used to communicate with other sensors and the central gateway of the WSN [ASS+14]. A list of sensor nodes currently available in the market can be found in this website [1]. WSN have widespread application and they are used in various scenarios such as military and civilian surveillance, structural monitoring,

[1] https://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes

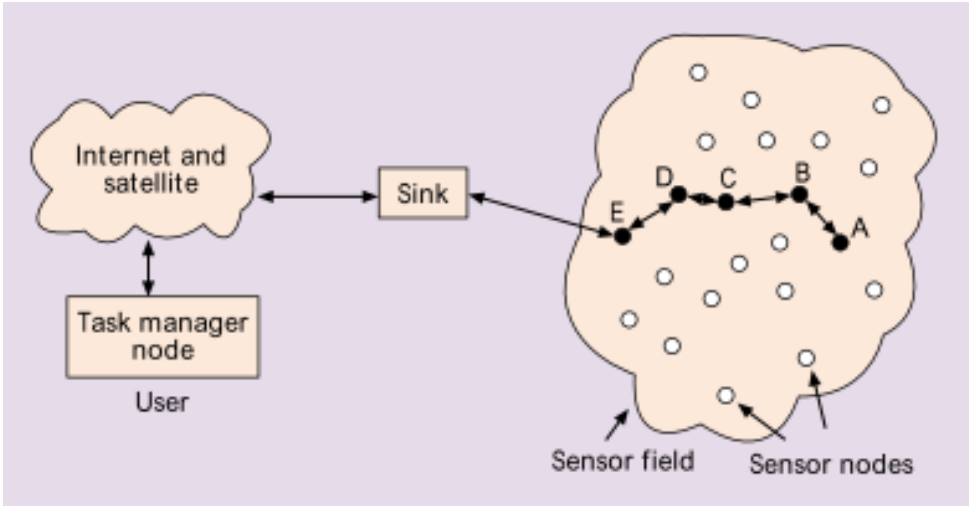


Figure 2.3: Typical WSN Architecture [ASSC02]

weather data collection, pollution level monitoring, patient monitoring and many others [FGN15].

2.4 Contiki OS

Contiki is one of the open source operating systems used in a network of resource constrained devices and IoT. It supports different hardware platforms, communication standards like IPv4, IPv6, and can run in extremely low-power devices. More information can be found from their official Contiki OS website [2].

2.5 Related Literature

Several studies have been carried out related to SDN and IoT. SDN has for years been mainly researched for its use in wired networks. However, aligned with the increase in the number of tiny wireless devices becoming part of the global network, there is a growing interest to study how SDN can be leveraged for network of resource-constrained devices like IoT, WSN, M2M and Personal Area Networks (PAN). Therefore, the main focus of this thesis and scope of the literature review will be on the use of SDN for small wireless-capable and resource-constrained devices, specifically WSN and IoT. Efforts have been made to bring both SDN and IoT, or SDN and WSN together to improve flexibility, performance, and other parameters. We can categorize the literature in this area into two groups. The first group of

[2] http://contiki-os.org

literature mainly focuses on architectural framework design and suggestion for SDN in WSN and IoT. Some of the studies in this category actually tried to practically implement their suggestions. The second group focuses on specific use cases of SDN in WSN or IoT, like smart management or energy efficient routing. We will next discuss some of the literature in these categories.

2.5.1 Architectural Frameworks

In this section we will discuss literature that specifically focus on architectural frameworks for deploying SDN in WSN, IoT and related networks.

One of the early studies that tried to integrate SDN to network of small wireless-capable and resource-constrained devices was carried out by Dely et al. [DKB11]. The authors made an architectural suggestion that integrates OpenFlow[MAB+08] in Wireless Mesh Networks (WMN) to enable flow-based forwarding and routing functionality. They further implemented a solution to handle the mobility of stations to show the feasibility of the architecture. The architecture they suggested is made up of two parts: Core Network and Stations. Core Network comprises a controller (e.g NOX [GKP+08]) and a Monitoring and Control Server. The controller is responsible for installing flow rules, making routing decisions and handling mobility. The Monitoring and Control Server queries information from mesh routers and clients to create a topology databases. The controller can interact with the Monitoring and Control Server to make the routing decisions and install flow rules. Station is the other component of their proposed architecture and it normally hosts a monitoring and control agent that communicates with the monitoring and control server and the controller to exchange information and build the topology. The work of this authors can be taken as a good first attempt to integrate SDN to WMN using OpenFlow as the main enabling protocol. This architecture, however, assumes that the nodes in the backbone of the WMN are normal computing devices and is, therefore, not suitable for networks of resource-constrained devices.

T. Luo et al. [LTQ12] also did a research in a similar topic and proposed an architectural framework for Software Defined WSN that creates a clear separation between data plane and control plane that uses Sensor OpenFlow (SOF), a customized communication protocol between the two planes. Their idea is to solve the inherent problems of traditional WSN like resource under-utilization, rigidity to policy changes and difficulty of management. They suggest that Software Defined WSN can transform the traditional WSN to be versatile, flexible, and easy to manage. They proposed some improvements to OpenFlow protocol such that it becomes compatible with the inherent properties of WSN, and they called this protocol Sensor OpenFlow (SOF). This paper has good architectural suggestion but it does not adequately explain how the proposed architecture and protocol can be implemented.

Costanzo et al. [CGMP12] further pursued the research in the area when they came up with the idea of Software Defined Wireless Networks (SDWN) that tries to bring SDN for use in wireless infrastructure-less networking scenarios like low rate, wireless personal area networks (LR-WPAN). Compared to the previous literature we discussed, the good thing about this work is that the authors extracted the requirements SDN should fulfill to be used in LR-WPAN and tried to implement them by developing a SDWN protocol stack. Some of the requirements they listed are flexible definition of flow rules, support for duty cycles to conserve energy and data aggregation. Their proposed architecture is composed of two nodes: sink node and generic node. The generic nodes implement a forwarding layer that forwards arriving packets according to the rules in the flow table that are installed by the controller. The sink node is similar to the generic node but has additional functionality. The controller is hosted in a more powerful embedded system or computer and is connected to the sink node using USB or some other communication interface. The controller is responsible for making network-wide decisions and installing forwarding rules in the generic nodes and sink node. The authors provided design and implementation details for collecting topology information, specifying and defining flow rules and actions, and the different packet formats. This can be considered as a first attempt to develop a custom protocol stack, other than OpenFlow, to be used in a network of resource-constrained devices. However, the proposed protocol only defines the messages that are exchanged between the generic nodes and sink nodes. It does not specify how the sink node should communicate with the connected controller, and hence can not be considered as a complete solution.

Han and Ren [HR14] also proposed what they call a novel WSN structure based on SDN. Their architecture is analogous to the SDN architecture and is made up of three components: master node, center node, and normal node. The master node is similar to the controller in SDN architecture and makes network-wide control functions like routing decisions, topology discovery. The center node is equivalent to the OpenFlow switch in SDN architecture and is responsible for matching and forwarding of packets according to the flow rules installed by the master node. The authors' approach to making analogy with SDN architecture is interesting but they did not explain how the center nodes communicate with the master node, what type of messages they use. There is no implementation provided and the architectural suggestion is mostly hypothetical.

Most of the architectural suggestions we have seen in the literature mainly considered flat structure. But Flauzac et al. [FGN15] came up with a new structured and hierarchical architecture using SDN in WSN, which they named it software-defined clustered sensor networks (SDCSN). In this architecture, the sensor network is divided into groups called clusters and each cluster has one node as a cluster head. The cluster head acts as an aggregator and coordinator to the cluster group. The

SDN controller is placed in the cluster head. A node can either be a Simple Node, Gateway Node or Cluster Head. Two clusters (domains) communicate through their Gateway Nodes. The SDN controllers in each of the cluster heads can be connected in Master/Slave mode or with equal status. Overall, this architectural suggestion tries to make a high-level design of how SDN can be organized in clustered and hierarchical structure in sensor networks. But it is not backed up with any simulated or real experimental setup and details of the protocols that are needed for the whole system to function are not discussed in detail.

Galluccio et al. [GMMP15b] recently came up with SDN solution for WSN, which they named SDN-WISE. Their approach tries to reduce the amount of information exchange between the sensor nodes and controller by making use of the state of each node (stateful approach). A detailed description of the prototype, the APIs, and implementation of the prototype are provided. Normal sensor nodes and sink nodes are the main components of the architecture. The sink node acts as a gateway between the normal sensor nodes in the data plane and the elements that made up the control plane. As the authors claim, and our literature review confirms, their work is the first real implementation of a custom OpenFlow like solution for WSN. Since this thesis will focus on testing the feasibility and evaluating the performance of SDN in real hardware network of small wireless-capable and resource-constrained devices, we will make use of SDN-WISE source code and try to extend it. More detailed discussion about SDN-WISE will be provided in section 2.6.

2.5.2 Use Cases

In this section, we will have a look at previous studies related to the use of SDN in WSN and/or IoT for specific use cases.

One of the early researches to deploy SDN in WSN was carried out by Mahmud and Rahmani in their published paper titled “Exploitation of OpenFlow in Wireless Sensor Networks” [MR11]. They suggest the use of OpenFlow technology to address reliability issue that is common in sensor networks. They argue that *flow-sensor*, which is an OpenFlow-based sensor, is more reliable than a normal sensor because the data packets, control packets and even the sensor nodes themselves can be easily monitored and routed whenever required. In their suggested architecture *flow-sensors* are used and the traffic is remotely controlled and the sensor nodes are also remotely monitored. The *flow-sensor* has two interfaces: control interface and data interface. The control interface uses OpenFlow protocol to communicate with the controller (access point) and exchange control packets. Whereas, the data interface (sensor buffer) uses TCP/IP connection with the controller (access point) and data packets are only exchanged in this channel. The authors tried to use some modeling tools to verify their concept and evaluate it. Despite claiming that SDN can improve the

reliability of wireless sensor networks, the authors did not show how exactly that can be achieved and their evaluation does not use reliability as a metric. They also did not provide practical implementation or future directions on how this can be materialized in real life scenarios.

A. De Gante et al. [DGAM14] also claim that smart management using SDN can solve some of the inherent problems in WSN and proposed a generic architectural framework to be used which implements the controller in the base station. They argue that the centralized intelligence of SDN can be used to find smart solutions for the management of the limited resources in sensor network like processing, memory, energy and communication capabilities. In their proposed architecture, they assume that the sensor network is made up of a base station and a number of sensor nodes. The sensor nodes do not make routing decisions, they rather forward packets using the rules that are installed by the base station controller. It is assumed that the controller will run in a special base station that has more power than the sensor nodes. The architecture of the controller is provided and it is composed of a middle-ware that has a mapping function for topology discovery and a flow-table definition consisting a database of flow rules for each node. This work can be considered of as more of theoretical because the authors did not substantiate their arguments with some kind of simulation or actual implementation. Their assumption that the base station of the sensor networks is more powerful and can host the controller does not always hold true.

2.5.3 Summary

As we have seen from the discussion of the literature, most of the researches carried out to date focus more on conceptual suggestion of frameworks that integrate SDN in network of resource-constrained devices. However, Galluccio et al. [GMMP15c] tried to implement their SDN-WISE framework and made their source code publicly available. Since the main objective of this thesis is to test the feasibility of SDN by deploying it in network of real hardware sensor nodes and to evaluate its performance, we decided to pursue our work based on SDN-WISE.

2.6 SDN-WISE

SDN-WISE is chosen as the SDN solution for network of resource-constrained devices, that our evaluation will be based on. The primary reason for the choice is the fact that the details of the solution are implemented and the source code is publicly available. In addition to that, the source code has been integrated with one of the popular operating systems in WSN, Contiki OS [DGV04]. This helps us to directly use the source code in real hardware and test the feasibility and evaluate

the performance of the SDN solution, which would make our results and analysis more realistic.

2.6.1 Overview

The authors define their solution as follows, "*SDN-WISE is a Software Defined Networking solution for WIreless SEnsor Networks. The aim of SDN-WISE is to simplify the management of the network, the development of novel applications, and the experimentation of new networking solutions*" [GMMPa]. As is stated in a presentation by the authors [GMMP15a], SDN-WISE tries to address the requirements of WSN like support for nodes with scarce resources, reducing energy consumption, and increase network flexibility. Their proposed solution uses logically centralized controller and flexible flow rule definition to make the network more robust and flexible. They also make use of duty cycles, data aggregation and stateful approach to reduce energy consumption. Duty cycle is a mechanism to save energy, where sensor nodes wake up only when there is data to send or receive, and go to sleep mode otherwise [Yan12].

2.6.2 SDN-WISE Protocol Architecture

As is common for Wireless Personal Area Networks (WPAN) like WSN, SDN-WISE protocol is based on the IEEE 802.15.4 Physical (PHY) and Media Access Control (MAC) layer specification. Three network elements are defined in the SDN-WISE architecture: Sink, Node, and Controller. The Sink is a gateway between the sensor Nodes and the Controller. All control packets should pass through the Sink to reach the Controller. Following the SDN terminology, the SDN-WISE protocol defines a control plane and data plane. As can be seen from Figure 2.4, the protocol stack of the data plane is implemented in the Sink and normal Node. The data plane is responsible for the forwarding of packets (FWD), local topology discovery (TD) and some in-network packet processing (INPP) like data aggregation. On the other hand, the control plane protocol stack is mainly implemented in the controller and partially on the Sink. The control plane implements specific network management logic or applications and takes care of the global topology of the network. The Sink and control plane implement an adaptation layer that helps them communicate. Equivalent to FlowVisor [SGY+09] in OpenFlow, SDN-WISE includes WISE-Visor which is a virtualization layer that enables different controllers with different policies to run over the same physical devices by abstracting the network resources.

2.6.3 SDN-WISE Protocol Details

The Sink and other normal Nodes implement three layers on top of the MAC layer as part of the data plane protocol stack.

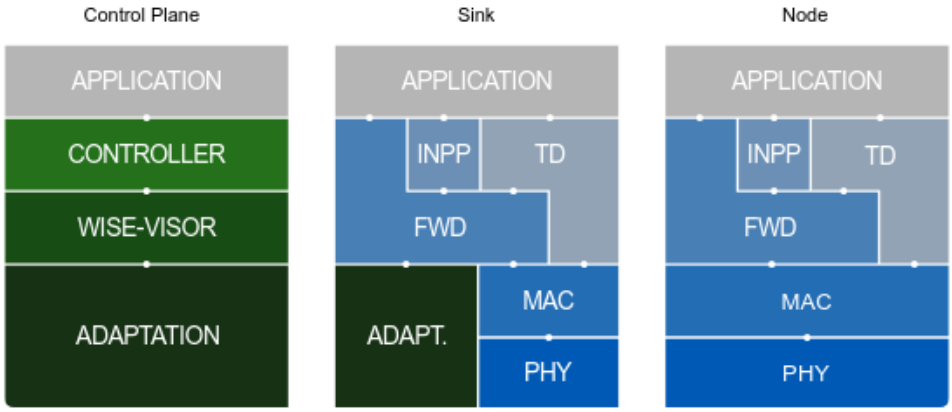


Figure 2.4: Protocol Architecture of SDN-WISE [GMMPa]

- **Forwarding Layer (FWD):** is responsible for handling incoming packets according to the rules specified in the WISE flow table, which is analogous to flow table in OpenFlow. It also keeps updating the WISE flow table according to the flow instructions sent from the controller.
- **Topology Discovery (TD):** this layer uses BEACON packets to establish neighborhood and create a table consisting the list of neighbors to a sensor node. It also calculates the best next path to the sink so that the sensor nodes can send control packets or reports to the controller via the sink.
- **In-Network Packet Processing (INPP):** this layer runs on top of the forwarding layer and performs some kind of data aggregation or other in-network processing tasks.

The Sink and control plane implement an **Adaptation Layer** that is used to format the messages exchanged between them in such a way that they can understand each other. **WISE-Visor** can be included in the control plane protocol stack to allow multiple controllers run on the same data plane network using abstraction and virtualization. One of the network management functions the **Controller** implements is Topology Management which is responsible to construct the global topology of the network by collecting reports from the Topology Discover of each sensor node.

Topology Discovery

This layer discovers local topology information of a specific sensor node and the shortest next-hop address to the sink. When the network starts, the sink nodes broadcast BEACON packets with their source address and the distance to the sink

set to 0. A sensor node A which receives this BEACON packet from the sink (node B) will perform the following things:

- Adds B to its neighbor table with its RSSI value, or updates its RSSI value if node B already exists in the neighbor table.
- If the existing distance to the sink is higher than the distance via node B, it updates the distance to the sink one plus the distance from B and sets node B as the next-hop to the sink.
- Node A in turn sends BEACON packets with the distance to sink node set, and broadcasts it to its neighbors.

In this way, each node maintains a local topology information and the shortest path to reach the sink node. This local topology information is reported to the controller via the sink node and the controller builds a global topology in the Topology Management module.

Packet Handling

The Forwarding Layer (FWD) is responsible for handling incoming SDN-WISE packets. The header of SDN-WISE packets is a fixed 10 bytes long and is made up of 7 fields as shown in Table 2.2. SDN-WISE defines 8 packet types:

- DATA Packet - this packet is just a variable size payload.
- BEACON Packet - this packet reports the distance of a source node from the sink and its battery level. It is a broadcast packet.
- REPORT Packet - reports the list of neighbors and is routed to the sink.
- REQUEST Packet - encapsulates a packet that has no match rule in the WISE flow table. It is sent to the controller via the sink.
- Response Packet - is a reply to flow request packet and contains a flow entry from the controller.
- OPEN-PATH Packet - is used to create a path between two nodes in the network.
- CONFIG Packet - used to read and/or write some configuration parameters.
- RegProxy Packet - is used to inform the control plane about the existence of a sink and some information about it.

Table 2.2: SDN-WISE Packet Header Fields [GMMPb]

Byte(s)	Name	Description
0	NET	Identifier of the network
1	LEN	Total length of the packet
2-3	DST	Destination address
4-5	SRC	Source address
6	TYP	Packet type
7	TTL	No. of hops remaining
8-9	NXH	Next hop address

Arriving packets are matched against the WISE flow table. As shown in Table 2.3, the flow table is composed of three sections: Matching rules, Actions and Statistics. Each entry in the flow table can have up to three matching conditions as part of the matching rule. Each matching condition has five fields: Operator, Size, State(S), Offset(Addr), and Value. Operator specifies the relational operator to be used against the Value. Offset and Size specify the starting byte and the number of bytes that should be considered starting from the Offset. If Size is 2 and Offset is 5, then two bytes starting from byte 5 are used to compare the relational operator with the Value. Each SDN-WISE has a WISE state array which contains the current state for each active controller. State(S) indicates whether the matching has to be done against the current packet or the state. IF S=0 the current packet is matched against the Value. Where as, if S=1 the state is matched. If all the matching conditions are satisfied, then the operation in the action part is carried out and the statistics is updated. The Action part of the flow entry is also composed of five fields: Type, M, S, Offset, Value. Some of the action types are Forward, Drop, Modify, ASK and others. When the action is Modify, S field specifies whether to modify the WISE state array or the current packet. M field specifies if only one matching entry has to be executed or not. After one successful flow entry is matched and the corresponding action is executed, if M=0 SDN-WISE stops browsing the flow table. However, it keeps searching for other matching rules if M=1. If no matching rule is found for the incoming packet, a request packet for flow entry is sent to the controller via the sink

Table 2.3: SDN-WISE Flow Table [GMMP15c]

Matching Rule					Action					Statistics	
Op.	Size	S	Addr.	Value	Type	M	S	Addr.	Value	TTL	Counter
=	1	1	0	0	Modify	1	1	0	1	122	23
=	1	1	0	1	Modify	1	1	0	0	122	120
-	0	-	-	-	Forward	0	0	0	D	122	143
-	0	-	-	-	Drop	0	0	-	-	100	42
-	0	-	-	-	Forward	0	0	0	D	100	32

2.6.4 SDN-WISE Limitations

A detailed discussion of the limitations of SDN-WISE will be covered in section 3.1. The following list gives an overview of some of the drawbacks (limitations) of SDN-WISE:

- The topology and neighbor discovery is not dynamic. Even if a node dies, the node still remains in the neighbor list table.
- The adaptation layer - which is the communication protocol between the sink node and the controller is not defined and implemented.
- The adaption layer is not fully implemented.
- Next-Hop to Sink (NXH) computing does not consider mobility into account.
- The authors only provided the source code of the sink and general node, but the SDN controller is not implemented.
- Even the source code for the sink and normal node is not complete. There are some functions which are not fully implemented like the handling of config packets and others.
- The developer community is passive, and there is a lack of complete and comprehensive source code documentation.

Chapter 3
Methodology and Testbed Implementation

The methodology chosen to carry out this research is mainly real hardware-based experimentation and measurement. The rationale behind the choice is so as undertake the evaluation in an environment that depicts reality and get reasonable results. In addition to that, supplementary methodologies are used in different phases of the project. The flow diagram in Fig. 3.1 shows the overall workflow and methodology used. The following are the main steps undertaken throughout the different phases of the research work:

- Literature Review: This was the first thing to do right after the commencement of the thesis work. Different research works related to the topic of our interest were reviewed and gaps identified. Among the available SDN solutions for a network of resource-constrained devices, SDN-WISE was chosen to be the basis for our research work and carry out the performance evaluation.
- Initial Setup and Gap Identification: In this step, an effort is made to set up the initial testbed using the source code provided by SDN-WISE. Details of how the solution works, the source code organization, the available functions, libraries, and its limitations are studied. Finally, the missing components and important features that have to be included in the testbed are listed out.
- Implementation: The next step was to implement the missing and required functionalities and/or components listed out in the previous step. The implementations are first tested and verified in a COOJA simulation environment before compiling and burning them into the real hardware. The COOJA simulator was very handy and helpful tool in debugging the implementations. It shows all the outputs of the nodes within one window, and it also allows us to move the nodes and create the desired topology. A whole section discussing the implementation contributions made by this thesis work will be covered in 3.2.
- Experimental Measurement: The verified and tested implementations are deployed into the actual hardware testbed to carry out the evaluation and

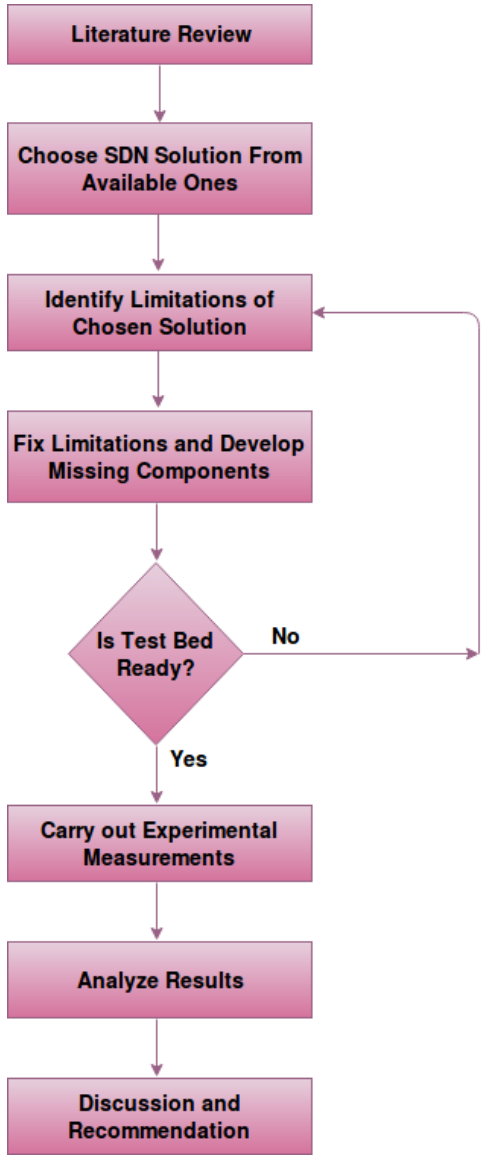


Figure 3.1: Thesis Methodology and Workflow

measurement of different performance metrics we set out. The output of each sensor node is saved into a log file as shown in Figure 3.2.

– Analysis: The log file collected from different measurements are analyzed and processed later on to extract the important metrics. The following six performance parameters are selected: packet delay, packet loss, convergence time,

signal overhead, rule installation time and mobility detection time. Different experimental scenarios are set out and their evaluation is carried out. Finally, the results obtained from the experimental measurement are used as a basis to make conclusions and future recommendations.

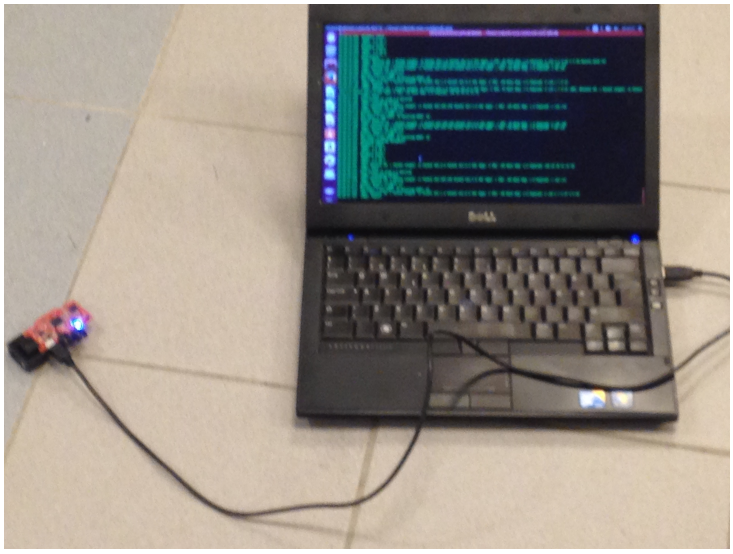


Figure 3.2: Data Collection in Action

3.1 Limitations of SDN-WISE Codebase

In this section, we will discuss some of the limitations we found out in SDN-WISE. For a complete SDN based WSN, three components are required: SDN Controller, Sink, and Normal nodes. SDN-WISE only implemented the code for Sink and Normal nodes. Further study of the original SDN-WISE source code also showed that it is not complete and additional implementation of missing functionalities and modification is required. We have identified the following limitations:

- The topology discovery was is not dynamic. If a node is dead or there is mobility, the nodes do not update their neighbor table and the dead node or displaced node is assumed as if it is still a neighbor. This limitation is modified and extended by adding expire timers and flags to check if the neighbors are still alive.
- The Serial communication callback function is not well implemented and it simply buffers incoming bytes. So it is necessary to improve the implementation to enable effective communication. Moreover, there needs to be a standard

(protocol) for communication between the SINK and the external controller. There are not yet standard messages developed like what is used in OpenFlow.

– The sensor nodes need to somehow reach the sink node so that the sensed data can be reported to the central device that would process the sensed data. This is achieved by computing the Next Hop to Sink (NXH) address from the received BEACON packets. Among the neighbor nodes, the node with the least number of hops from the sink node is taken as the NXH address to the sink. However, because of the lack of robustness of the topology discovery SDN-WISE uses, the NXH computing also lacks some robustness. Once a node is assigned as the NXH address, it is not updated even if the NXH node has died because of battery or moved away from the neighborhood. This poses a serious problem as the nodes can not reach the sink node with a dead node assigned as their NXH address.

– The other most important thing was the lack of SDN controller. The SDN-WISE solution does not actually recommend any controller. It is up to the developer to develop the controller. So to make our testbed complete and do our evaluation and experimentation, we needed to develop the controller. We develop a simple controller using Python that implements the Dijkstra algorithm. We make use of a very important python library called *networknx* that helps us build the topology graph and use its shortest path algorithms like Dijkstra.

3.2 Experimental Testbed Implementation

In this section, a discussion of the main implementation contributions made to set up the complete testbed are presented.

3.2.1 Initial Setup

The sensor nodes we will use for our experimental setup are Z1 motes (further discussion about Z1 motes is found in section 4.2). The original SDN-WISE source code is written for, and assumes, Tmote Sky motes [1] as the target hardware. Therefore, slight modifications in the initial set up of the source code is needed to make it work smoothly on the Z1 motes. The first thing is to change the target of the compilation from 'sky' to 'z1'. The other most important change is to correctly configure the Universal Asynchronous Receiver/Transmitter (UART). Embedded systems and sensor nodes, in particular, require a means of communicating to the external world for various reasons like transmitting data to/from another device, sending and/or receiving of commands, or for debugging purposes [Msp]. One of the most common

[1] http://tmote-sky.blogspot.no/2013/11/the-tmote-sky-platform.html

ways to achieve this is using UART. The UART controller normally configures the baud-rate and input callback function of the serial input. The UART controller ensures that both ends of the communication are configured correctly. The main task of the UART is to transmit bytes as a sequence of bits. The initial attempt to compile the original source code results in a compilation error related to the UART configuration. There are two ways to configure the UART: as **UART0** or **UART1**. The original SDN-WISE source code uses **UART1** initialization since their target hardware is Tmote Sky motes. However, Z1 motes use **UART0** initialization. Therefore, we replaced the header file *uart1.h* by *uart0.h* in the *sdn-wise.c* main program of SDN-WISE, and also replaced the initialization functions *uart1_init* and *uart1_set_input* into *uart0_init* and *uart0_set_input*.

3.2.2 SDN-WISE Serial Communication

As previously discussed, the serial communication is used for debugging purposes, or to send data and commands to/from the sensor nodes. The *uart_rx_callback* function is responsible for handling incoming bytes from the serial port. The original implementation of the callback function in SDN-WISE simply buffers the incoming bytes until the buffer is filled. It then sends a signal that the buffer is full and data is ready. To effectively interact with the sensor nodes and make them programmable and configurable, we defined our own specific commands that can be interpreted by the sensor nodes. All the commands from a computer to the sensor nodes are transmitted over the serial communication via the USB. Considering our specific requirements and commands we want to have, it was decided to make the length of the commands to be 5 bytes, followed by '\n' as the end of command marker. Table 3.1 summarizes the list of serial commands defined, and they are described below:

- **d**xxyy - This command instructs a given sensor node to generate a data packet with source address x+1.0 and destination address y+1.0. Example: d1122 command will generate data packet with source 2.0 and destination 3.0
- xx**u**yy - This command instructs a sensor node with address x+1.0 to install a unicast forwarding flow entry for packets matching with destination address y+1.0. Example: 11u33 command will install a flow-table entry in sensor node 2.0 that says any packet with destination address of 4.0 should be forwarded in unicast to node 4.0.
- xx**u**yz - This is similar to the previous command. It installs a flow entry in node x+1.0 that says any packet with destination address y+1.0 should be forwarded in unicast to node z+1.0. Example: 22u53 command will install flow-table entry in node 3.0 that says any packet with destination address 6.0 should be forwarded to node 4.0 as unicast.

- xx**b**yy - This command instructs a sensor node with address x+1.0 to install a flow entry that says broadcast all packets with destination address y+1.0. Example: 11b33 command installs a flow-table entry in sensor node with address 2.0, a flow entry that broadcasts all packets with destination address of 4.0
- xx**d**yy - This command also instructs a sensor node with address x+1.0 to add a flow entry that says drop all packets with destination address y+1.0. Example: 33d44 command instructs node 4.0 to install flow entry that says drop all packets with destination address 5.0
- xx**s**fx - This command instructs sensor node with address x+1.0 to show (display) its flow-table. Example: 22sf2 will display flow-table of sensor node with address 3.0
- xx**r**fx - This command instructs sensor node with address x+1.0 to remove (clear) its flow-table. Example: 11rf1 will clear the flow-table of node with address 2.0.

Serial Command	Example	Description
dxxyy	d1122	Generate data packet with src address 2.0 and dst address 3.0
xxuyy	11u22	Install unicast flow entry in node 2.0; flow match dst address 3.0 unicast to 3.0
xxuyz	22u32	Install unicast flow entry in node 3.0; flow match dst address 4.0 unicast to 3.0
xxbyy	33b22	Install broadcast flow entry in node 4.0; flow match dst address 3.0 broadcast
xxdyy	11d44	Install drop packet flow entry in node 2.0, flow match dst address 5.0 drop
xxsfx	55sf5	Show flow table of node 6.0
xxrfx	33rf3	Remove (clear) flow table of node 4.0

Table 3.1: Summary of Serial Commands Defined

In the implementation of the *uart_rx_callback* function of the serial communication, a buffer of size MAX_PACKET_LENGTH 116 bytes long is used. Incoming packets are stored in the buffer. The first five bytes from the serial line are copied from the buffer to a temporary buffer. As shown in flow diagram in Figure 3.3, the callback function reads the next five bytes after each '\n' marker and then checks if it is one of the defined commands. According to the specified command, a data packet or configuration packet is created and sent to the specified destination. The improved implementation of the callback function enables the serial communication line to read commands and process them accordingly. This is the basis for communicating and programming the sensor nodes.

3.2.3 SDN-WISE Extension - Communication Sink to Controller

SDN-WISE defines the following packet types: BEACON, DATA, REPORT, REQUEST, CONFIG, and OPEN-PATH. These packet types are used when the sensor

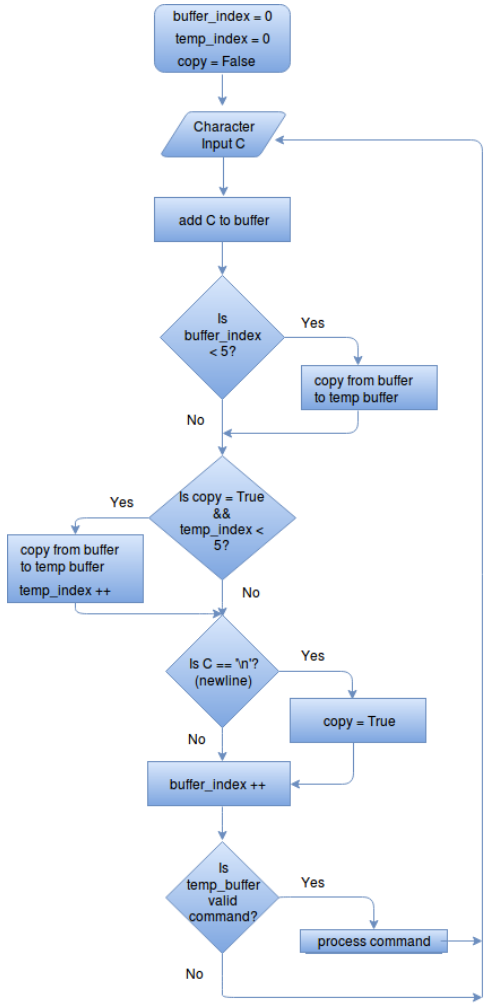


Figure 3.3: Flow Diagram of the Serial Communication Callback Function

nodes communicate with each other. However, SDN-WISE has not implemented any sort of communication protocol between the sink node and the external SDN controller that is installed on a PC. Therefore, we implemented our own custom serial communication-based protocol and mechanism that helps the sink node and the SDN controller to talk to each other. The two most common messages that are sent from the sink node to the SDN controller are REQUEST for flow entries and REPORT of local topology information. In the other way round, from the SDN controller to the sink node, CONFIG messages and commands are sent frequently. In the protocols we defined, a local topology information is sent to the SDN controller by prepending REPORT: marker to the message, followed by the topology information in the next

line. The same is true with a request for flow entry message. REQUEST: marker is prepended followed by the source and destination of the request. Figure 3.4 shows an example of how the REQUEST and REPORT messages are constructed. The REQUEST message in the figure is requesting for flow entries to route a packet from node 2.0 to node 6.0. The REPORT message is sent from node 3.0, and it is advertising its local topology information stating that node 1.0, 2.0 and 4.0 are its neighbors with 80, 82, and 86 RSSI values respectively. The SDN controller then reads the messages line by line. If the marker is REPORT, it reads the next line to get the topology information. Similarly, if the marker is REQUEST, it reads the next line to get the source and destination for the request. The REPORT and REQUEST messages are handled by the SDN controller. One of the configuration serial commands discussed in 3.2.2 are used as RESPONSE to the REQUEST messages.

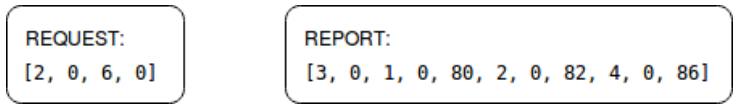


Figure 3.4: REQUEST and REPORT Message Formats

3.2.4 SDN-WISE Extension - Dynamic Topology Discovery

SDN-WISE uses periodic BEACON packets to establish neighborhood between nodes. If one sensor node receives BEACON packet from another node, it adds the other node into its neighbor table with its address and RSSI value. However, the original implementation SDN-WISE for neighbor discovery is not dynamic. Once a node is added into the neighbor table, it still remains there even if the node is dead, or if it has moved away and the topology has changed due to mobility. Due to the resource constrained nature of sensor nodes, some of the nodes might run out of battery. Mobility is also an inherent property of networks. Therefore, SDN-WISE should be robust enough to learn the change of topology and update its view of the network. To this end, we modified the data structure of the neighbor-table of SDN-WISE and added a new *is_alive* flag that enables us to keep track of the dynamics in the topology. When a BEACON packet is received from a specific node, the node is added into the neighbor-table for the first time and the *is_alive* flag is set to 1. Consecutive BEACON packets from the same node will keep the *is_alive* flag set to 1. SDN-WISE uses period of 5 seconds to send BEACON packets. If BEACON packets are not received, which indicates the node is dead or has moved away, then the *is_alive* flag is reset after 15 seconds (3*BEACON period). We wait for three times the period of the BEACON period before resetting the *is_alive* flag. The node is then removed from the neighbor-table 10 seconds after the *is_alive* has been reset. The flow diagram in Figure 3.5 describes how the dynamic topology works.

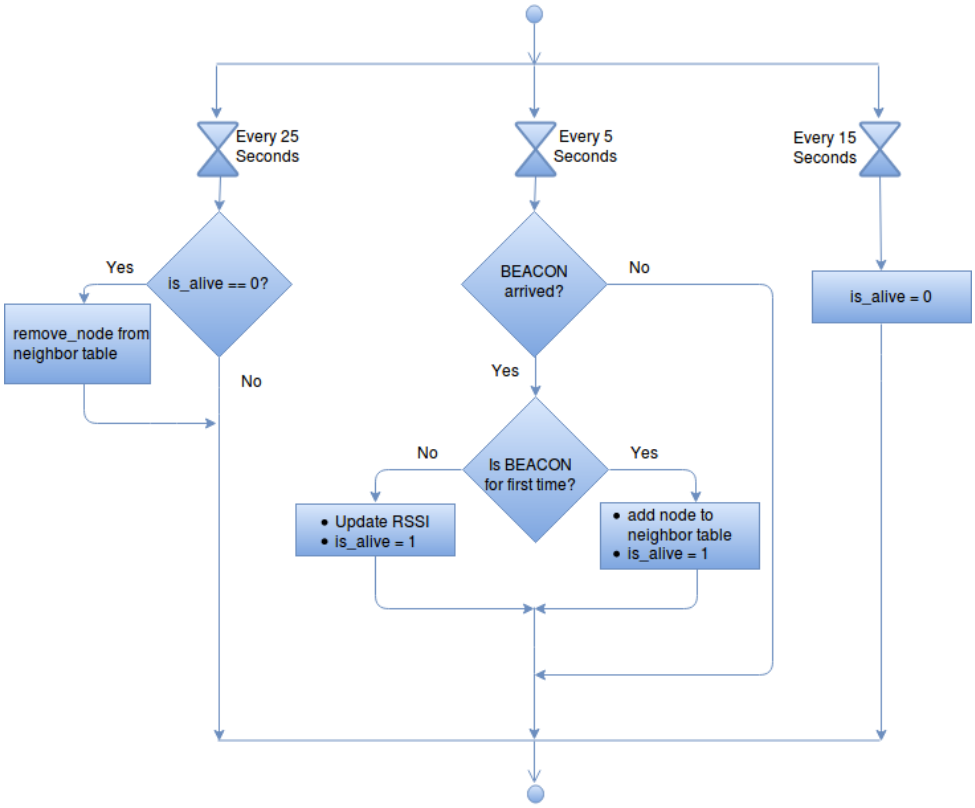


Figure 3.5: Flow Diagram of How Dynamic Topology Discovery Works

3.2.5 SDN-WISE Extension - Robust Next Hop to Sink

Each sensor node has to be able to reach the sink node so as to report and send sensed data. To this end, each node has a Next Hop to Sink (NXH) address that points towards the neighboring next node that has the shortest path to the sink node with strongest RSSI value. Similar to the problem we discussed earlier for the neighbor discovery, the Next Hop to Sink assignment also lacks robustness. Once the address of a specific neighbor is set as the NXH, it is not updated even if the node is dead or has moved away from its neighborhood. Our experimental work showed that this is a very serious issue. This is because of the fact that the NXH will point towards a non-existing node, since the NXH is not updated after the node is dead or removed. This prohibits the sensor nodes from reaching the sink node, which technically makes the sensor network dysfunctional. This problem can be solved by instructing the node to recomputes its NXH if the existing NXH is removed from neighbor-table due to mobility. This gives a chance for the node to receive incoming BEACONs and compute its new NXH.

3.2.6 SDN-WISE Extension - Packet Handler

Each sensor node requires a packet handler function that processes the serial communication commands mentioned in section 3.2.2. We modified the packet-handler function of SDN-WISE so that it can interpret and execute the commands we defined. An illustrative flow diagram of the modified packet-handler is shown in Figure 3.6.

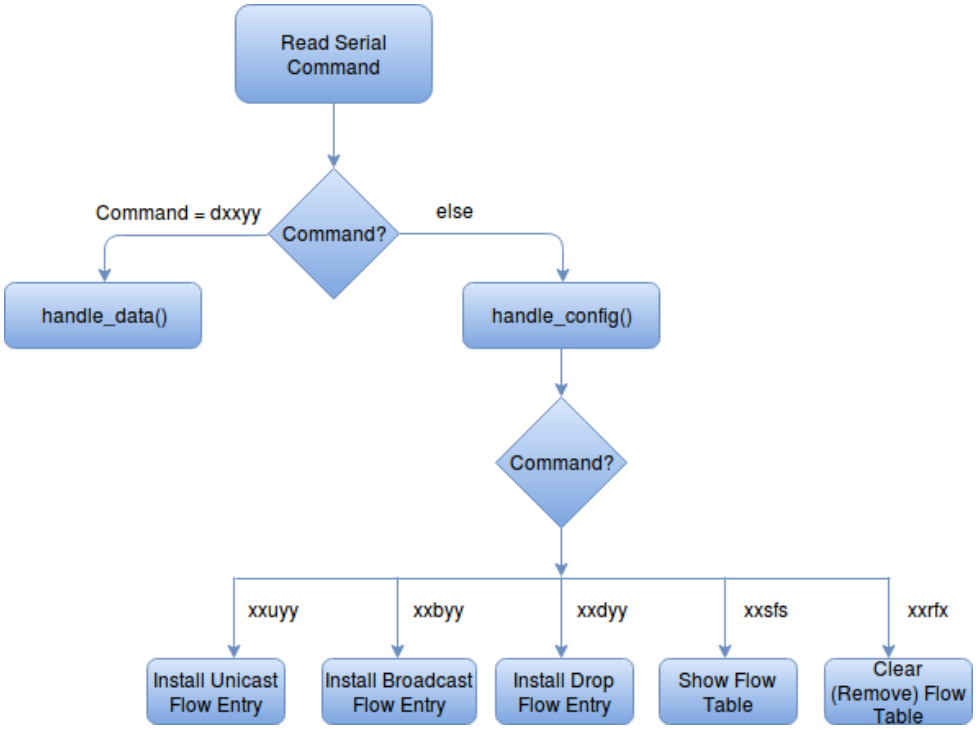


Figure 3.6: Flow Diagram of Modified Packet-handler

3.2.7 SDN-WISE Controller - PY-SDN-WISE

Sensor nodes that use SDN-WISE can be manually configured using the commands we discussed in subsection 3.2.2. The flow table of each sensor node can be populated in a proactive manner. This can be preferable when we have a small number of nodes in the sensor network. However, there need to be very intelligent and carefully crafted applications in the SDN controller to manage complex sensor networks. We have implemented our own Python-based SDN controller to make our testbed automated and get realistic measurements. We used the *threading* [2], *serial* [3] and *networkx* [4] Python libraries to implement our controller. The package management system for Python (pip) is used to install these libraries. We have to install pip first to install the other packages. The following Linux commands are used to install the packages:

```
sudo apt-get install python-pip
sudo pip install --upgrade pip
sudo python -m pip install pyserial
sudo python -m pip install networkx
```

Two type of messages are sent from the sensor nodes to the controller: REPORT and REQUEST. As previously discussed in 3.2.3, REPORT message contains an array of local topology information. Where as, the REQUEST message contains source and destination address of the path to be set up. The SDN controller we developed handles these messages to build the topology from the REPORT messages, and responds to REQUEST messages using the shortest path from the topology. Figure 3.7 shows a flow diagram of how the messages are processed. The SDN controller keeps reading messages from the serial interface line by line. If the message is REPORT, it extracts the information inside the message and uses it to construct the whole topology. In a similar manner, if the message is flow REQUEST, it extracts the source and destination of requested path. It then computes the shortest path from source to destination using Dijkstra's algorithm and replies with RESPONSE message, if a path exists.

[2] https://docs.python.org/3/library/threading.html
[3] https://pythonhosted.org/pyserial/
[4] https://networkx.github.io/

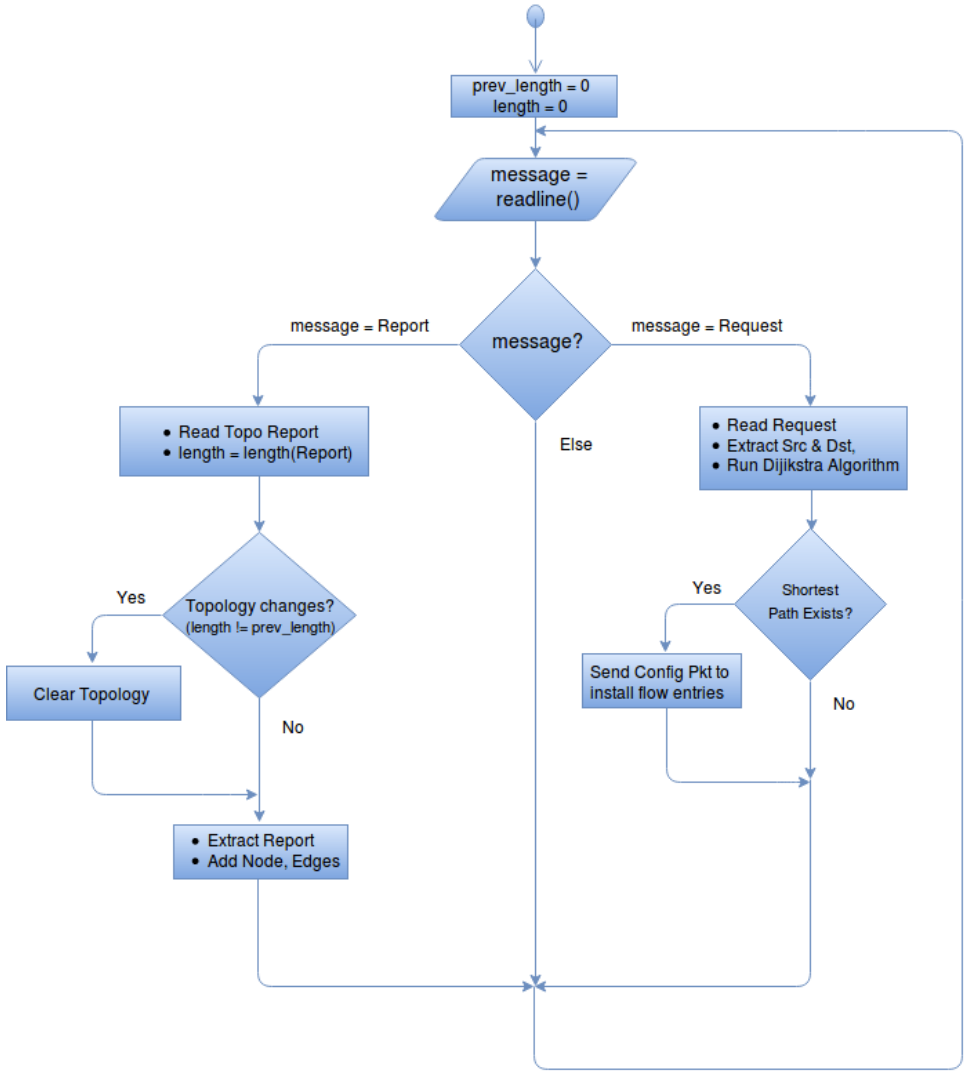


Figure 3.7: Flow Diagram of How SDN Controller Handles SINK Node Messages

The activity diagram in Figure 3.8 shows how the nodes communicate with the controller to establish a route by requesting flow entries. When Node 1 wants to send a data packet to Node 2, it first sends flow REQUEST message to the SDN Controller via the sink node. The controller then sends a RESPONSE back to Node 1 via sink. However, since sink node does not have a flow rule for how to forward packets to Node 1, it sends flow REQUEST to the controller. Flow RESPONSE is sent to sink nodes. Since the flow RESPONSE to the previous flow REQUEST

from Node 1 has not reached back to Node 1, Node 1 sends a new flow REQUEST again. This time the flow RESPONSE reaches to Node 1 and the corresponding flow rule is installed. Similar steps, but with more message exchanges, are used when sending data packets from Node 1 to Nodes 3, 4, and 5. The source code of the SDN controller implementation is given in Appendix A.1.

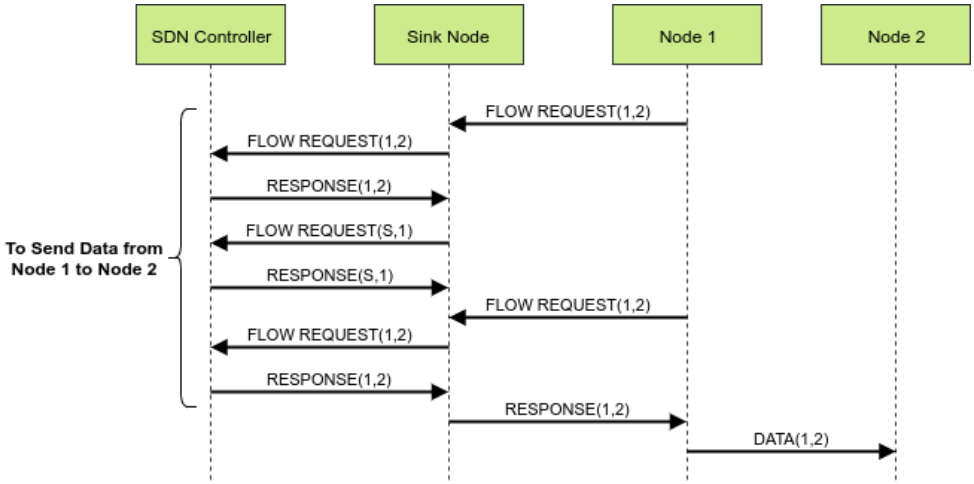


Figure 3.8: Nodes and Controller Establishing Route in Single Hop Mesh Topology

Chapter 4
Experiment Setup

After all the software components that would make the experimental testbed complete are implemented, the next step was to evaluate its performance by considering different scenarios. Commonly used performance metrics like packet delay, packet loss, convergence time, signal overhead, rule activation time and mobility detection time are evaluated. These metrics have been re-defined in a way to fit our specific context. A total of four scenarios with varying topology are used for the evaluation. In the first three scenarios, the focus is to evaluate the performance of the testbed under different topologies. In the fourth scenario emphasis is given to investigate how the dynamic mobility handling mechanism we implemented works and to measure its associated metrics.

Since the performance evaluation is mainly time-related, each event that occurs in the sensor nodes and the SDN-WISE controller is time stamped to make the measurement possible. For each experiment carried out, the log of the sender node, the receiver node, and the SDN-WISE controller is saved to a file. These logged files are later processed to extract the performance metrics values.

4.1 Performance Metrics

In the following sections the metrics considered in the performance experiments are defined.

4.1.1 Convergence Time

In our case, convergence time refers to the time taken for the SDN-WISE controller to discover the whole topology. Due to the unstable radio links and environmental interference, sensor networks can at times show unpredictable behavior and the network might not converge quickly. All the sensor nodes are started at the same time to calculate the convergence time at the SDN controller. The convergence time is measured with topology report period set to 20 seconds, which means each

node reports its local topology every 20 seconds. Beacon packets which are used to establish neighborhood between sensor nodes are broadcasted every 5 seconds by each node.

4.1.2 Packet Delay

Packet delay is the time it takes for a packet to reach from source node to its destination node. The delay considered in our case includes the propagation delay and the processing time of SDN-WISE related functions in each node along the path of the packet. This can be measured by calculating the time-stamp difference between when the packet is generated and when it is received by the destination node. Since synchronization is important when measuring delay, we used internet synchronized timers between the two computers that will log the output of the sensor nodes in both ends. However, it should be noted that its is difficult to achieve absolute synchronization.

4.1.3 Packet Loss

Packet loss refers to the number of packets that fail to reach their destination due to network congestion, interference, or when a path is not established yet. Due to the limited memory and other resources of sensor nodes, buffering of packets is not a good option in WSN. So whenever the route (path) of a packet is not established yet or whenever there is network congestion, packets might be lost. This can easily be measured by assigning an ID or counter to each packet in the source node and then checking which and how many of the packet arrive in the destination node.

4.1.4 Control Message Overhead

Control message overhead refers to the number of SDN-WISE specific control messages that are exchanged between the sensor nodes, or between sensor node and SDN-WISE controller. These messages include *local topology report* messages, flow request and response messages. Beacon messages are not included as overhead because they are not only SDN-WISE specific messages. They are used in other WSN protocols also.

4.1.5 Rule Activation Time

Rule activation time is the time it takes for a flow to be installed. This is measured by calculating the time difference between the instant where a flow request is made until the flow rule is installed in the target.

4.1.6 Mobility Detection Time

This is the time taken for the SDN controller to learn that a specific node has moved or removed (is dead). This can be calculated as the time difference between the instant a given sensor node is removed from the network and the time the controller learns about its removal.

4.2 Hardware

The sensor nodes used for the testbed are Z1 motes which are manufactured by Zolertia. According to its datasheet [Zol10], Z1 is a low-power wireless module that is compliant with IEEE 802.15.4. It is commonly used as a platform for testing and research in the WSN world. It uses an MSP430 family 16-bit MCU (microcontroller unit) and CC2420 transceivers by Texas instruments. It has 8 kB RAM and a 92 kB Flash memory, and its CC2420 transceiver has an effective data rate of 250 kbps and operates at 2.4 GHz [LAB+16]. Z1 motes can be battery powered or USB powered by connecting a cable through their micro-USB connector which can also be used for debugging. Z1 motes support some of the most commonly used open source operating systems in the WSN community like TinyOS [1] and Contiki OS [2]. The available SDN-WISE source code is integrated with the Contiki OS.

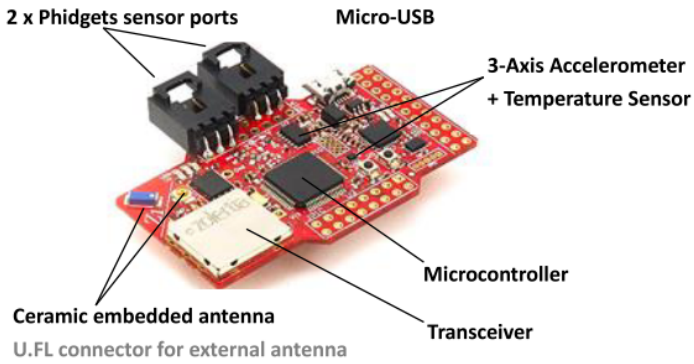


Figure 4.1: Z1 Sensor Node (Mote) [Yan12]

4.3 Software

All the sensor nodes in the network will be installed with SDN-WISE which is integrated with Contiki OS. The SDN-WISE source code is compiled both for sink

[1] https://github.com/tinyos/tinyos-main
[2] http://contiki-os.org

node and other normal nodes. One sensor node will be installed with the sink SDN-WISE and the rest are installed with normal SDN-WISE. As mentioned in the experimental testbed implementation, both a custom Python-based SDN controller and a script to generate data packets have been developed.

4.3.1 Toolchain to Compile Source Code

A set of toolchain is required to compile the SDN-WISE source code and upload (burn) it into the Z1 hardware motes. Following the instructions from [LAB+16], the following toolchain dependencies are installed in our 32 bit Ubuntu (Linux) machines:

```
sudo apt-get update
sudo apt-get install gcc-arm-none-eabi gdb-arm-none-eabi
sudo apt-get install build-essential automake gettext
sudo apt-get install curl graphviz unzip wget
sudo apt-get install gcc gcc-msp430
```

Each application or project in Contiki needs to have a corresponding Makefile to be compiled correctly. SDN-WISE uses the following Makefile which can be modified according to what specifications we want.

```
CONTIKI_PROJECT = sdn-wise
all: $(CONTIKI_PROJECT)
SMALL = 1
CONTIKI = ../contiki
CONTIKI_WITH_RIME = 1
PROJECT_SOURCEFILES = flowtable.c packet-buffer.c address.c neighbor-table.c
    packet-handler.c node-conf.c packet-creator.c
TARGETDIRS += ../targets
CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"
include $(CONTIKI)/Makefile.include
```

As is specified in the above Makefile, each project can include a project specific configurations which can be saved in a file like project-conf.h. A sample project-conf.h file is shown below.

```
#ifndef PROJECT_CONF_H_
#define PROJECT_CONF_H_
#define BATTERY_ENABLED 0
#define DEBUG_SINK 1
#define SINK 1
#define NODE1 0
#define NETSTACK_CONF_MAC csma_driver
#define NETSTACK_CONF_RDC contikimac_driverr
#endif
```

The bash script in 4.1 which is provided as part of the SDN-WISE source code, with some modification, is used to compile the code. After the source code is compiled, it

Algorithm 4.1 Bash Script to Compile SDN-WISE Source Code

```
#!/bin/bash
TARGET="z1"
make TARGET=$TARGET DEFINE=SINK=1
mv sdn-wise.$TARGET compiled/sink.$TARGET
make TARGET=$TARGET DEFINE=SINK=0
mv sdn-wise.$TARGET compiled/node.$TARGET
```

is then uploaded (burned) into the Z1 hardware. The good thing about Z1 motes is that the compiled code can be flashed (burned) into the hardware using its mini-USB port by connecting it with a computer. Assuming the compiled code is saved as node.z1 in compiled/ folder, The following command is used to upload (burn) the code:

```
sudo make compiled/node.upload MOTES=/dev/ttyUSB0
```

We can provide inputs or display the output of the sensor nodes by connecting to the serial port. There are two possible ways to do so.

```
sudo make login MOTES=/dev/ttyUSB0
sudo make serialdump MOTES=/dev/ttyUSB0
```

If the Ubuntu OS is 64 bit one, the following additional libraries need to be installed to make the serialdump and login work.

```
sudo apt-get install lib32z1 lib32ncurses5
```

Both login and serialdump use the Contiki application that is located in ../contiki/-tools/sky/serialdump.c. However, serialdump provides the timestamp of the different events displayed in the output, where as login only displays the output without timestamp.

4.3.2 Script to Generate Packets

The simple python script shown in 4.2 is developed to generate data packets that will be used to measure the performance metrics of the testbed. As we have discussed in the implementation of the experimental testbed, most of the interaction between the sensor nodes and the computer that hosts the SDN-Controller or other scripts is carried out over the serial communication via the USB. So the packet generating script iterates and writes 'd1122\n' to the opened serial line. The sensor node

will then interpret this as a command to generate data packet with source address 2.0 and destination address 3.0. More about the serial commands can be found in subsection 3.2.2. The newline character '\n' is a delimiter for the serial communication. It signals the end of one command and beginning of next one.

Algorithm 4.2 Python Script to Generate Packets

```
#!/usr/bin/env python
import time
import serial
print("Python Data Packet Generator")
ser = serial.Serial('/dev/ttyUSB0')
for x in range(1000): #assuming 1000 data packets are to be generated
        sendcommand = 'd'
        sendcommand += str(1)
        sendcommand += str(1)
        sendcommand += str(2)
        sendcommand += str(2)
        sendcommand += str('\n')
        send = bytearray(sendcommand)
        ser.write(send)
        time.sleep(5)
```

4.4 Evaluation Scenarios

The evaluation is carried out by considering different scenarios with varying topology and when there is mobility. Some initial experimentation and playing around with the SDN-WISE codebase shows that the behaviour of the network depends on its topology. A total of four scenarios with varying topology are chosen. The first three scenarios focus on the performance of the testbed when topology is varying. The last scenario focuses on how mobility is handled by the dynamic topology discovery we implemented, and the last .

4.4.1 Scenario 1: Singe Hop Mesh Topology

As shown in Fig. 4.2, a single hop mesh topology where each sensor node is placed at one hop distance from the sink node and all the other nodes is used for performance evaluation in this scenario. This is the simplest kind of topology in WSN.

4.4.2 Scenario 2: Multi-Hop Topology 1

The topology shown in 4.3 is used for this scenario. Since the data packets that are used for the measurement are generated from Node 1, the topology in this scenario will have destination nodes with hop length of 1, 2, and 3.

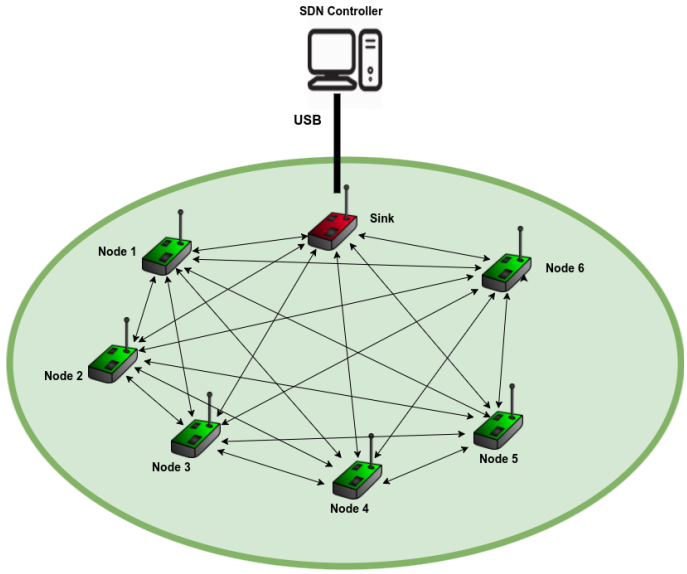


Figure 4.2: Scenario 1: Single Hop Mesh Topology

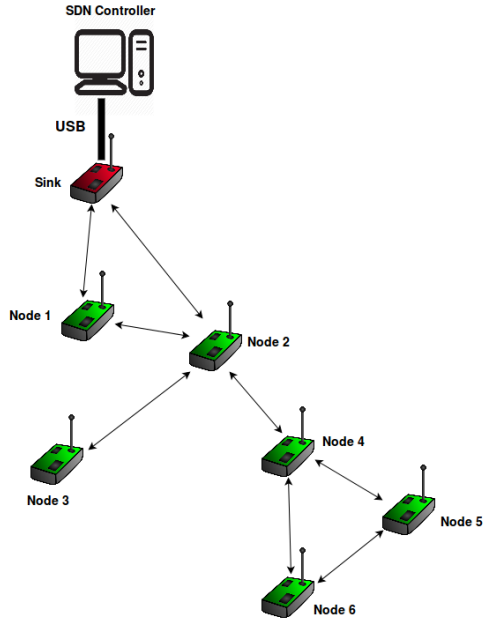


Figure 4.3: Scenario 2: Multi-Hop Topology 1

4.4.3 Scenario 3: Multi-Hop Topology 2

The topology used in this scenario, as shown in Figure 4.4, is similar to the previous one except that we have destination nodes up to hop length of 4 from Node 1. The previous scenario had destination nodes with hop lengths 1, 2, and 3, while this scenario has hop lengths of 1, 2, 3, and 4. Table 4.1 shows the hop distance of each node in the first three scenarios. The results of both scenarios will be combined to analyze effect of hop length on the packet delay and other performance metrics.

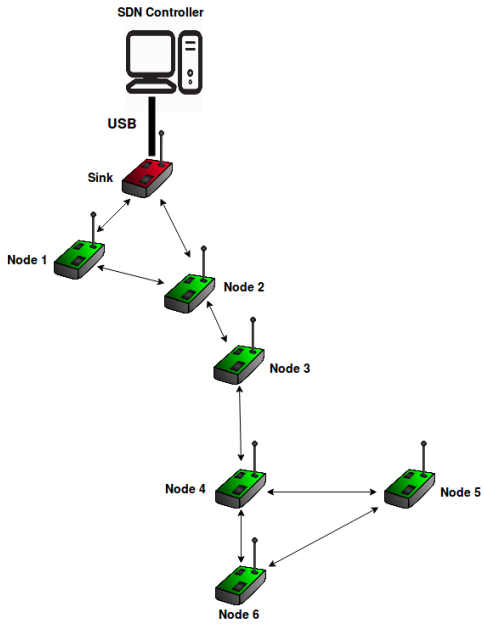


Figure 4.4: Scenario 3: Multi Hop Topology 2

	1 Hop Distance	2 Hop Distance	3 Hop Distance	4 Hop Distance
Scenario 1 (Single Hop Mesh Topology)	Sink, Node 2, 3, 4, 5, 6	-	-	-
Scenario 2 (Multi Hop Topology 1)	Sink, Node 2	Node 3, 4	Node 5, 6	-
Scenario 3 (Multi Hop Topology 2)	Sink, Node 2	Node 3	Node 4	Node 5, 6

Table 4.1: Hop Distance of Sensor Nodes from Node 1

Experiment Execution

The experiment execution procedures for scenarios 1, 2, and 3 are the same. We think that the packet loss and control message overhead will depend on the intensity of the packets being generated. To this end, we will do the performance evaluation when packets are generated at 1 second interval and at 5 second interval.

- **Convergence Time:** In the implementation of SDN-WISE, the sink node activates the other normal nodes. So, to measure the convergence time, all the sensor nodes are reset first. Then the sink node is started and it activates all the other sensor nodes. Both the sink node and the SDN controller are started at the same time. The convergence time is measured from the time the sink node starts until the SDN controller learns the topology of the whole network. The time stamped output of the sink node and the controller are saved into a file and the time difference is measured from that log file.

- **Packet Delay, Packet Loss, Control Message Overhead, Rule Activation Time:** The same procedure is used to measure these performance metrics. The packet generating script is used to generate data packets from Node 1 to each of the other nodes (Node 2 - Node 6). Time instants where each packet is sent and received are logged both at the sender and receiver ends. The difference is then calculated to determine the packet delay. Each generated packet is assigned an ID and sent with the packet as payload. We then determine the loss by looking at the packets that managed to arrive at the destination node. When data packets are generated, Node 1 will request the SDN controller for flow entry to route the packets. The instant where these requests arrive in the SDN controller and the instant where the corresponding flow entry is added in Node 1 are both logged into a file and the time difference is calculated to determine rule activation time. The control message overhead is calculated by counting the number of flow request, response and report messages that are exchanged between the sink node and the SDN controller with in a given time span.

- **Mobility Detection Time:** Nodes 2 to 6 are each removed from the topology. The mobility detection time is then the time difference from the instant the node is disconnected until the SDN controller learns the new topology.

4.4.4 Scenario 4: Mobility Handling

In this scenario, the topology in Fig. 4.5 is used to investigate how the dynamic topology discovery of the testbed works.

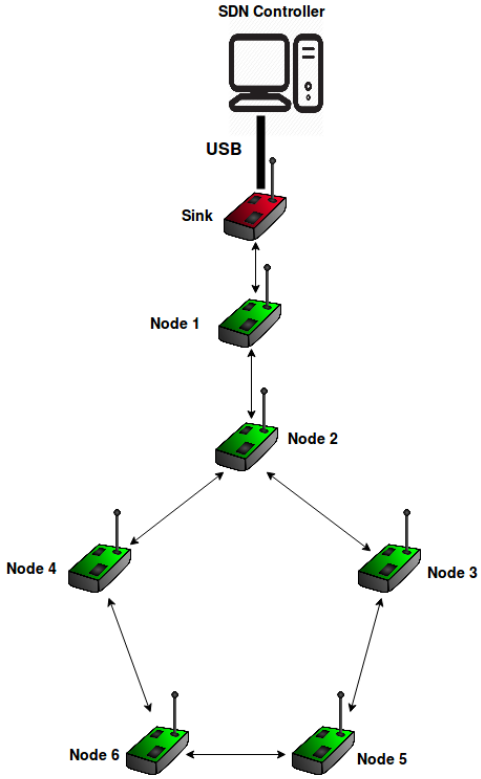


Figure 4.5: Scenario 4: Before Mobility Occurs

Experiment Execution

The packet generating script is used to generate data packets from Node 1 to Node 6. Initially, the sensor nodes does not have flow rules to forward the packets, and thus send flow requests to the controller. The SDN controller will choose the shortest path and installs flow rules to route the data packets accordingly. In the middle of the data transmission, at a random instant, Node 4 is removed from the topology. The response of the dynamic topology discovery mechanism of the testbed and how it is handled is investigated in this scenario.

Chapter 5
Results

The results obtained from the experimental measurements are presented in this chapter. Further analysis and discussion of the results will follow in the next chapter.

Setting up the experimental test bed is one of the biggest achievements of this thesis work. The testbed implements SDN paradigm in a network of small wireless-capable and resource-constrained devices. We made sure that every component of the testbed is working correctly prior to doing performance measurements. The main focus in the first three scenarios of our experimental setup was to measure the performance of the testbed under different topologies. In these scenarios, measurements for convergence time, packet delay, packet loss, control message overhead, rule activation time and mobility detection time are carried out. However, the fourth scenario focuses on how mobility is handled by the testbed, and how long it takes for the network to repair itself and re-establish a new route. The results obtained are presented in the following sections. The error bars in the graphs correspond to a 95% confidence interval. All the measured data from the experiments is attached in Appendix B.

5.1 Scenario 1 to 3

The experimental procedures for scenario 1, 2 and 3 are similar, and the results obtained from performance measurement of these three scenarios are presented next.

5.1.1 Convergence Time

Convergence time is one of the metrics we used for the performance evaluation. Networks have to discover their topology prior to making any routing decisions. The convergence time measured in our testbed is the time it takes for the SDN controller to learn about the whole topology. The convergence time measurement experiment is repeated 20 times. In each run, the SDN controller and all the sensor nodes are started at the same time and the instant when all the topology is discovered is recorded. Figure 5.1 shows the convergence time distribution for the scenarios 1,

2, and 3. The result indicates that the single hop mesh topology converges faster

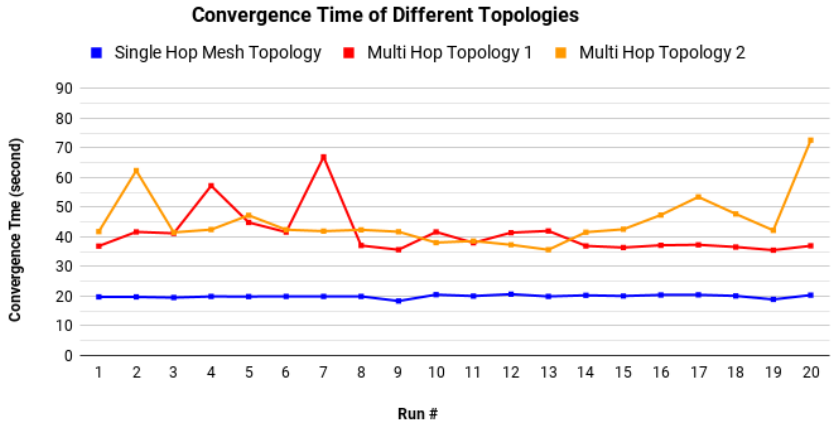
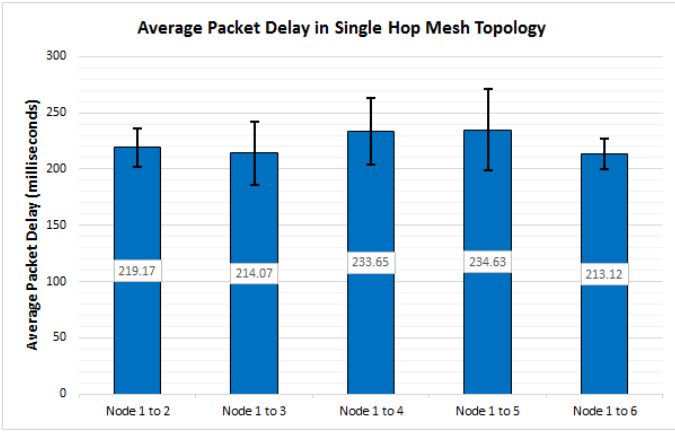


Figure 5.1: Convergence Time of Different Topologies

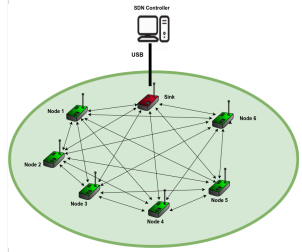
than the multi-hop topologies. Considering the fact that *local topology report* is sent every 20 seconds, the convergence time is expected to be equal to or greater than 20 seconds. However, some of the convergence time results we got are slightly less than that. This is partly due to measurement error and poor synchronization between the timers. All the sensor nodes and the SDN controller have to be started at exactly the same time to get accurate convergence time measurement. However, the SDN controller program sometimes fails to start as intended when it tries to access the serial line. The serial line (USB) is sometimes occupied by other programs running in the background and gives error output when the controller tries to access it. This creates some delay to the starting time of the controller and the report packets seem to have arrived earlier, from the time reference of the controller.

5.1.2 Packet Delay

Using the script we developed, data packets were generated from Node 1 to each of the other 5 nodes. The result in Figure 5.2 shows the average packet delay in scenario 1 (single hop mesh topology). Since each node is at one hop distance from the packet generating Node 1, their average packet delay is close to each other. In scenarios 2 and 3, however, the sensor nodes are at different hop distances and the results in Figures 5.3 and 5.4 reflect this fact. The average packet delay of nodes at the same hop distance are close to each other. As the number of hops from the source node increases, the average packet delay also keeps increasing.

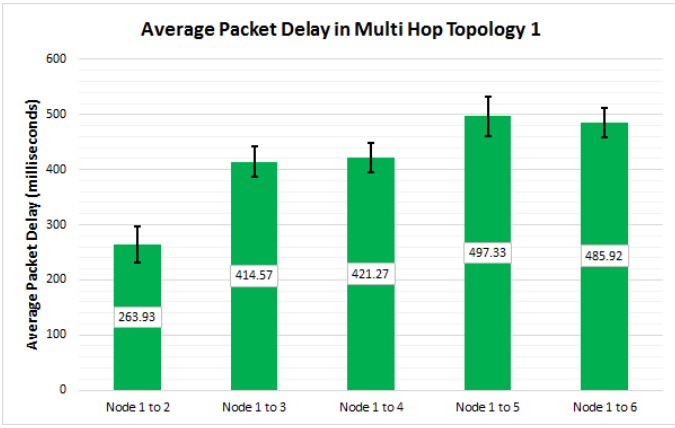


(a) Average Packet Delay

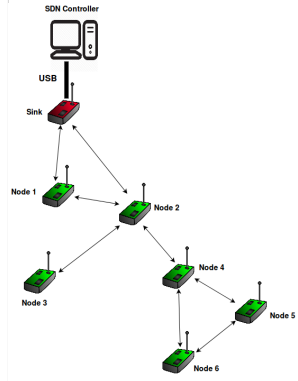


(b) Scenario 1

Figure 5.2: Scenario 1: Single Hop Mesh Topology Average Packet Delay



(a) Average Packet Delay

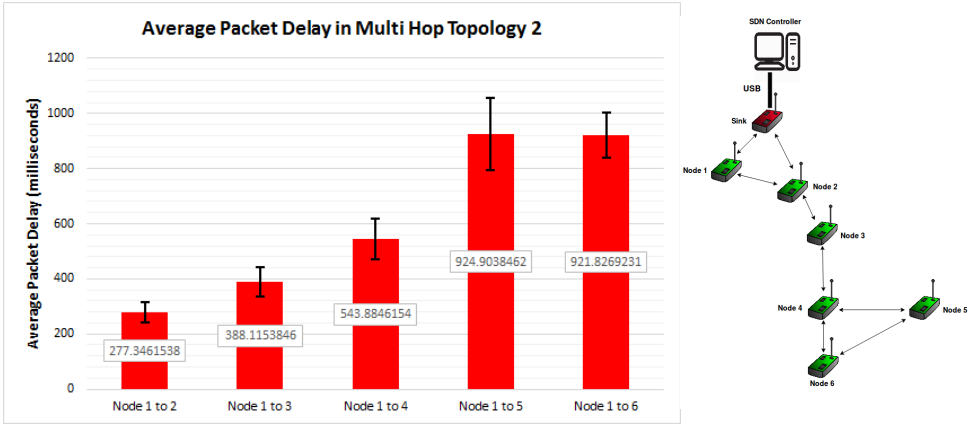


(b) Scenario 2

Figure 5.3: Scenario 2: Multi-Hop Topology 1 Average Packet Delay

5.1.3 Packet Loss

Packet loss occurs when a path between the source and destination nodes is not yet established, or due to the unstable nature of the radio links. Since the loss caused by the latter case is common to all WSN, the focus of our evaluation has been on the loss caused by the former case i.e loss occurred before routes are established. The data packets used for measuring the loss are generated at a 5 second interval after the whole topology is discovered (converged). Figure 5.5 shows the packet loss distribution in scenario 1 (single hop mesh topology). Most of the data points have

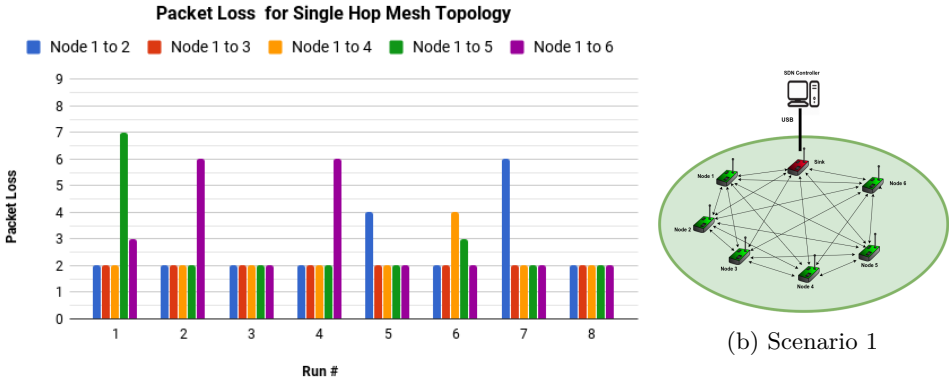


(a) Average Packet Delay

(b) Scenario 3

Figure 5.4: Scenario 3: Multi-Hop Topology 3 Average Packet Delay

a pattern with 2 packet being lost. However, due to the unstable nature of the radio link between the sensor nodes, a higher packet loss than expected can sometimes occur. The topology in scenario 1 consists of nodes at single hop distance from each



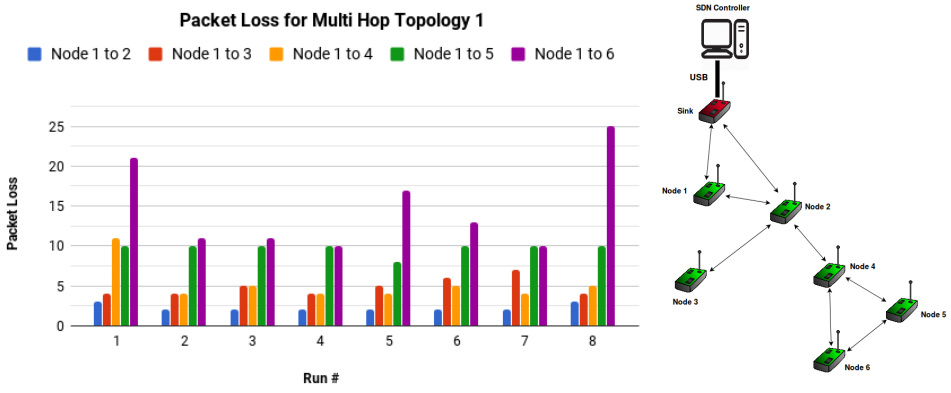
(a) Packet Loss Distribution

(b) Scenario 1

Figure 5.5: Scenario 1: Packet Loss Distribution

other. So, it takes a shorter time to establish a path for routing packets, and the packet loss is minimum. On the other hand, the packet loss in multi-hop topologies (scenario 2 and 3) is higher, because it takes more time to establish a path between the intermediate nodes. The results obtained reflect this fact, as shown in Figures 5.6 and 5.7. Nodes at the same distance have more or less approximate packet loss values, except where the radio links fluctuate and cause more packet loss. At normal

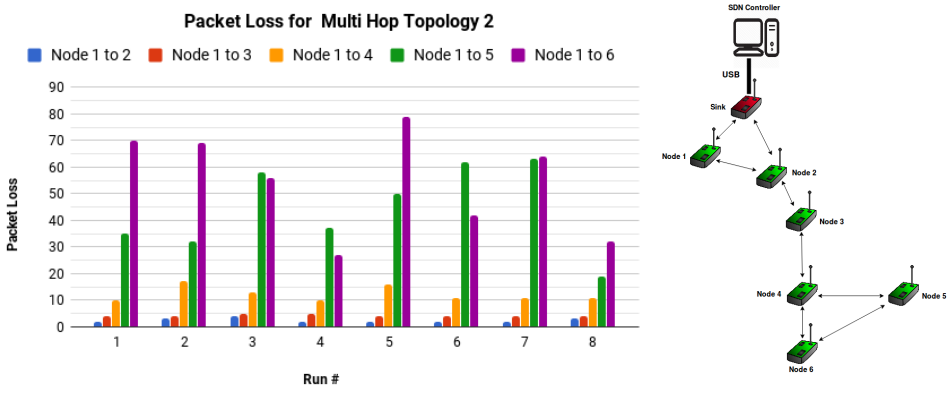
conditions, the packet loss increases as the number of hops increase. It takes more time to establish a long path with multiple hops, and packets are dropped until the path is established.



(a) Packet Loss Distribution

(b) Scenario 2

Figure 5.6: Scenario 2: Packet Loss Distribution



(a) Packet Loss Distribution

(b) Scenario 3

Figure 5.7: Scenario 3: Packet Loss Distribution

5.1.4 Control Message Overhead

Control messages include the periodic topology reports by each sensor node, and also request and response messages for flow entries. Since the *local topology reports* are periodic messages, their overhead is averaged per minutes. As shown in Figure 5.8,

an average of close to 20 topology report messages are sent every minute to the SDN controller.

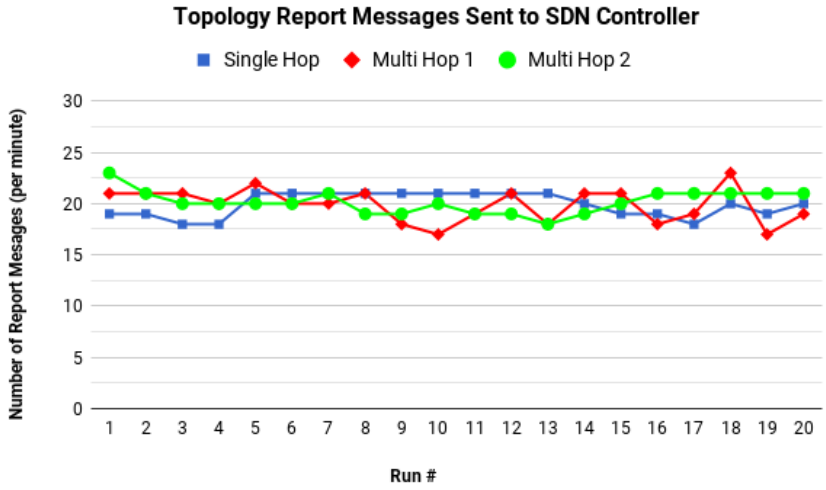
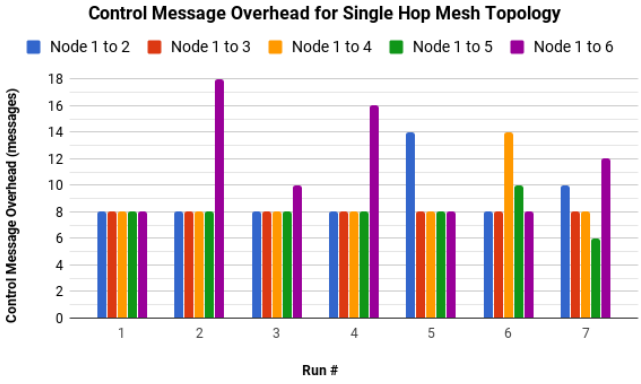
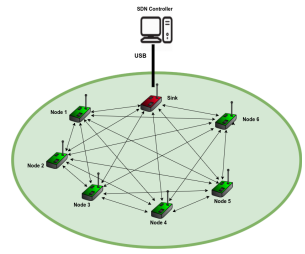


Figure 5.8: Number of Report Messages Sent to SDN Controller Every Minute

The other control message overheads that have been given more focus in this study are flow request and response messages. Each node sends these messages to the controller whenever there is no matching entry in their flow-table. Figure 5.9 shows the control message overhead (*flow request* and *response*) in scenario 1. Some kind



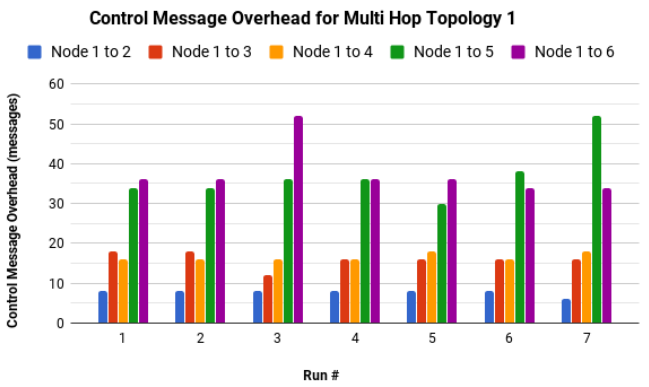
(a) Control Message Overhead Distribution



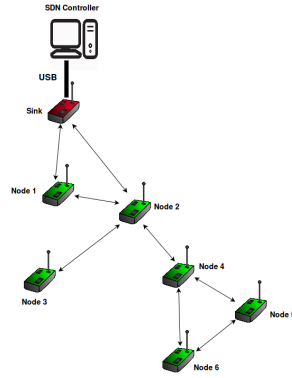
(b) Scenario 1

Figure 5.9: Scenario 1: Control Message Overhead Distribution

of uniformity is observed, with most of the data points having an overhead of 8 messages. The few variations are attributed to the fluctuating topology caused by the unstable nature of the radio links. When the topology fluctuates, the SDN controller can not correctly determine the shortest path and install the flow rules. As a result, the sensor nodes keep sending flow request messages until a flow rule is installed and the overhead increases. Figures 5.10 and 5.11 show the control message overhead in scenarios 2 and 3 respectively. As the number of hops increases, the intermediate nodes generate flow requests to set up a path, and this leads to the increase in control message overhead.

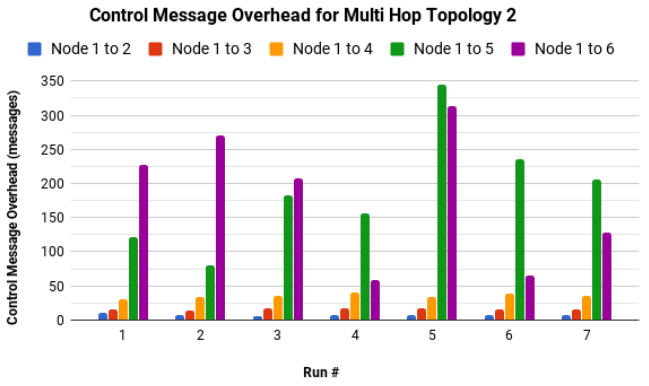


(a) Control Message Overhead Distribution

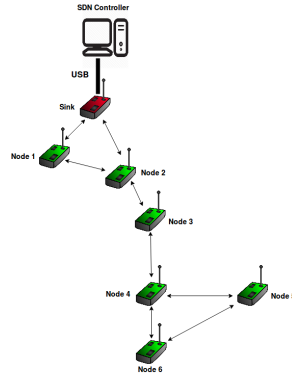


(b) Scenario 2

Figure 5.10: Scenario 2: Control Message Overhead Distribution



(a) Control Message Overhead Distribution



(b) Scenario 3

Figure 5.11: Scenario 3: Control Message Overhead Distribution

5.1.5 Rule Activation Time

All the data packets that are used for the measurement are generated from Node 1. Experimental measurement was carried out to evaluate how long it takes to install flow rules. In this context, flow rule activation (installation) time is the time taken from the instant a flow request is made until a flow entry (rule) is installed in Node 1. The result in Figure 5.12 shows the distribution of flow rule activation (installation) time. The measurement was repeated 30 times. In average, it takes around 193 milliseconds to install flow rules in Node 1.

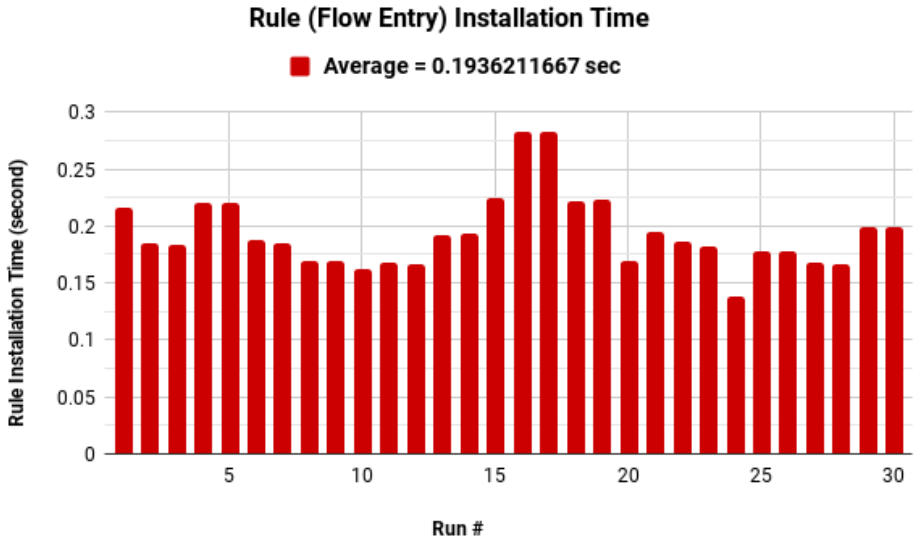


Figure 5.12: Rule (Flow Entry) Installation Time

5.1.6 Mobility Detection Time

The SDN controller learns the dynamic change in the topology of the testbed through the *local topology report* messages. These reports are sent by each sensor node every 20 second interval. In this experiment, Nodes 2 to 6 were each removed from the topology, at a random instant, to investigate how long it takes for the testbed to detect the mobility. Figure 5.13 shows the average mobility detection time when Node 2, 3, 4, 5, and 6 are removed from the topologies in scenario 1 (single hop mesh topology), scenario 2 (multi-hop topology 1) and scenario 3 (multi-hop topology 2). The results show that the mobility is detected with in one to three reporting period, i.e approximately within 20 to 60 seconds. In addition to that, mobility in single hop mesh topology is detected faster than in the multi-hop topologies.

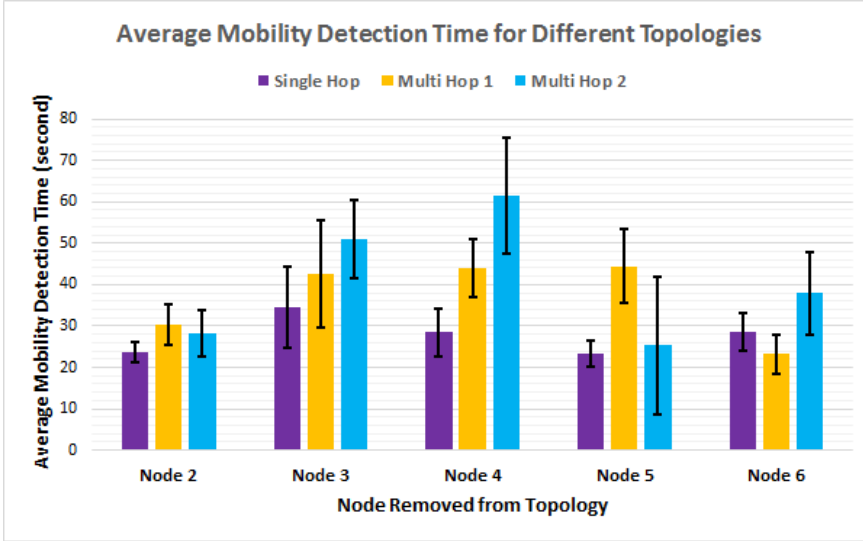
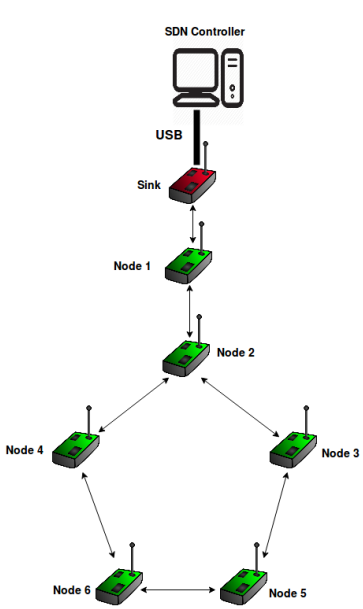


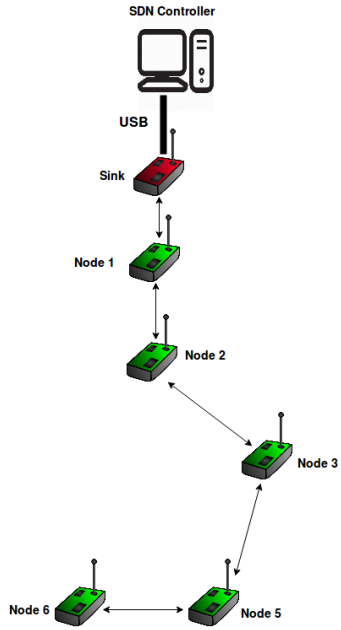
Figure 5.13: Average Mobility Detection Time for Different Topologies

5.2 Scenario 4: Mobility Handling

In this scenario, we wanted to investigate how the testbed handles mobility and change of topology. Initially, the topology in Figure 5.14a is used and the packet generating script generates data packets from Node 1 to Node 6. The controller establishes the path and the data packets start to be routed from Node 1 to Node 6 via Node 4 i.e Node 1 -> Node 2 -> Node 4 -> Node 6. As the data transmission continues, Node 4 was removed from the network at random instants. The controller then detects the mobility and tries to re-establish a new path. The result of 18 repeated experiments shows that the controller detects the removal of Node 4 after approximately 30 seconds in average. After the controller detects the removal, it resets the flow-table of Node 2 so as to establish a new rules and routes. Node 2, then, sends a new flow request message to set the route for data packets destined to Node 6. The controller responds to the requests and installs new flow rules that routes the data packets following the path Node 1 -> Node 2 -> Node 3 -> Node 5 -> Node 6, as shown in Figure 5.14b. The repair time of the network is the sum of the mobility detection time and the time to establish the new path. Figure 5.15 shows the distribution of repair time after the mobility occurs.



(a) Scenario 4: Before Mobility Occurs



(b) Scenario 4: After Mobility Occurs

Figure 5.14: Scenario 4: Topology Before and After Mobility

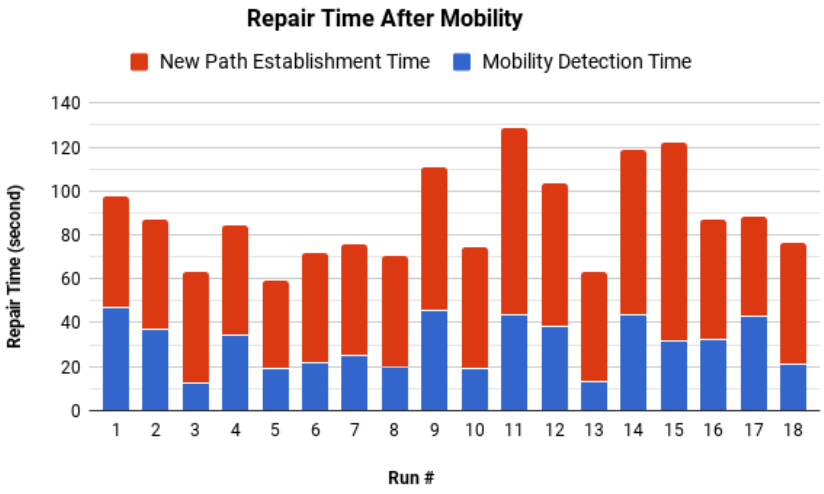


Figure 5.15: Repair Time Distribution After Mobility

The results obtained from our measurements show that the average repair time is around 87.87 seconds, out of which 57.7 of it is the average time to set up the new path, as depicted in Figure 5.16. However, it should be noted that the time it takes to establish a new path depends on the topology and hop distance of the path to be established. The longer the path, the longer time it takes to establish the new path.

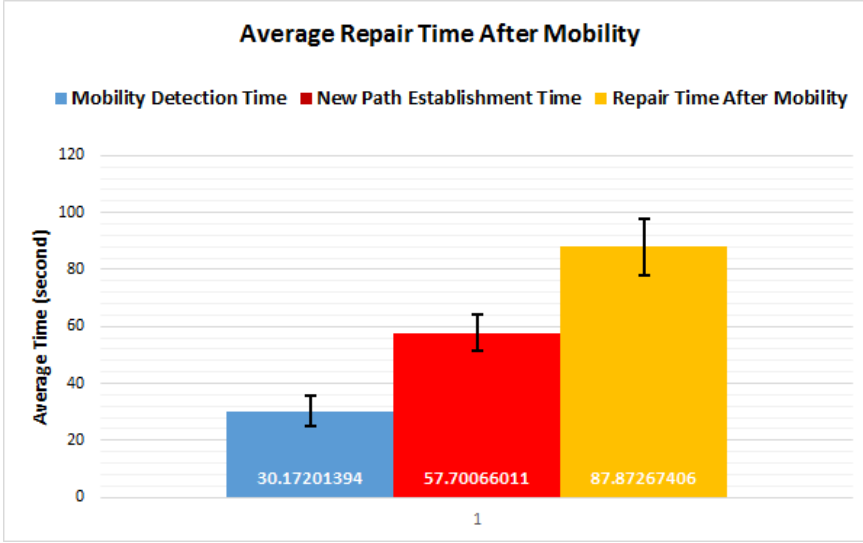


Figure 5.16: Average Repair Time After Mobility

Chapter 6
Discussion

Setting up the experimental testbed was one of the main objectives of this thesis work. The performance measurements of the testbed has been presented in the previous chapter. In this chapter, we will discuss some of the important trends and pattern observed from the performance evaluation. The limitations of this thesis work will also be discussed briefly.

6.1 Effect of Topology on Convergence Time

Three different topologies were used to study their impact on the performance of the testbed. The *local topology report* messages, that are sent every 20 seconds by each sensor node, are used by the SDN controller to construct the whole topology of the network. As the average result in Figure 6.1 shows, a single hop mesh topology converges much faster than the multi-hop topologies that span larger geographic area. This indicates that as the topology grows, the convergence time increases as well.

Convergence time of WSN was also studied by Ali [Ali12]. Ali carried out the evaluation in a COOJA simulation environment where RPL [Int] is used as the routing protocol in the WSN. The average convergence time they achieved was around 19 seconds. The results we obtained from our experimental measurement for convergence time range from 19.95 to 45s. Considering the fact that their measurement is carried out in a simulated environment and ours is in a real environment, it can be said that the performance of SDN is adequate.

6.2 Effect of Hop Distance

The sensor nodes in WSN might be placed at different hop distance from each other. To analyze how this affects the performance, the topologies used in the experimental scenario were set up to have nodes with hop distance of 1, 2, 3 and 4. As shown in Figure 6.2, the average packet delay increases as the number of hops increases. The

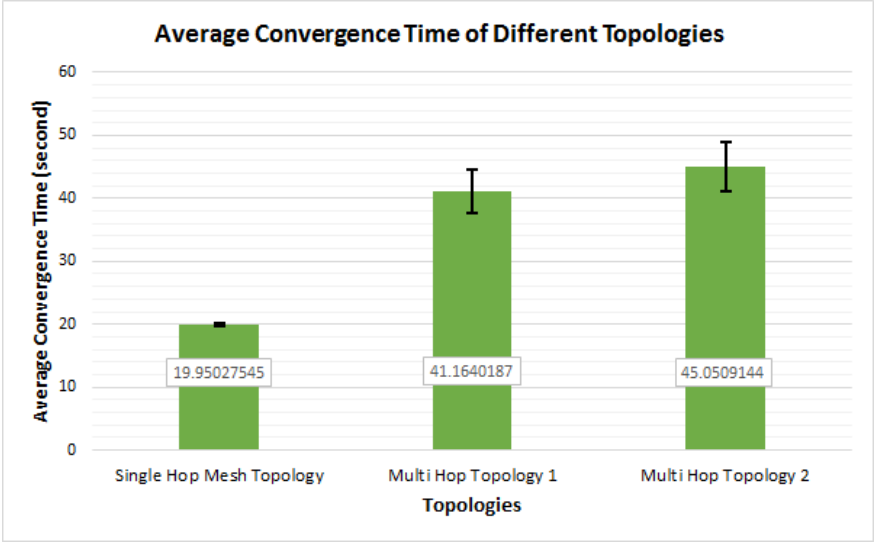


Figure 6.1: Average Convergence Time of Different Topologies

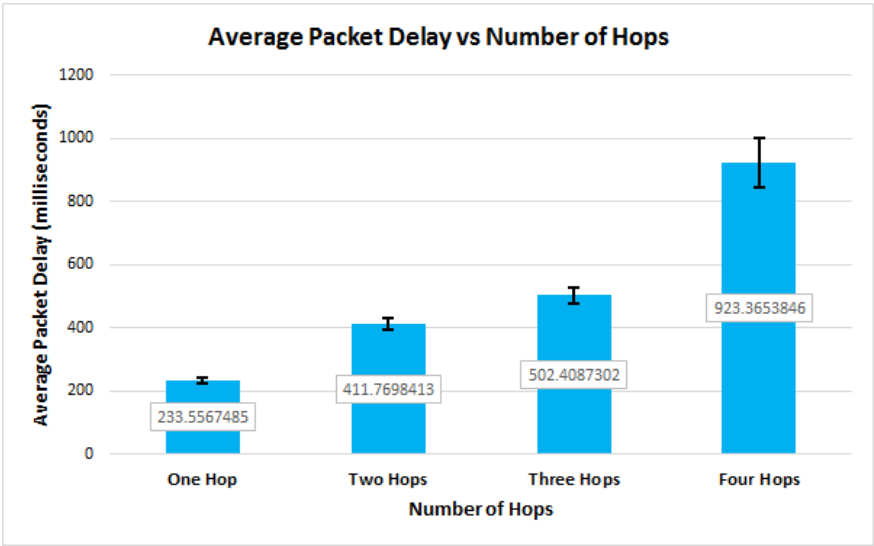


Figure 6.2: Average Packet Delay vs Number of Hops

packet loss and control message overheads are also affected by the hop distance. If a path between two nodes has multiple intermediate nodes, then these nodes have to send more flow entry requests to the controller so as to establish the path. Since this takes more time and the packets are dropped until the path is established, the packet loss increases as well. Figures 6.3 and 6.4 show this pattern of increase in

packet loss and control message overhead as the number of hops increases.

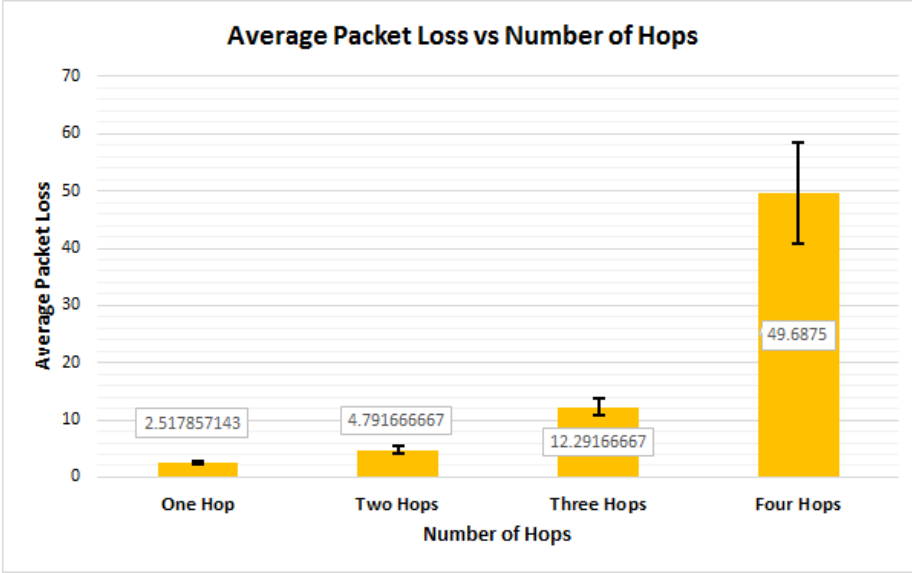


Figure 6.3: Average Packet Loss vs Number of Hops

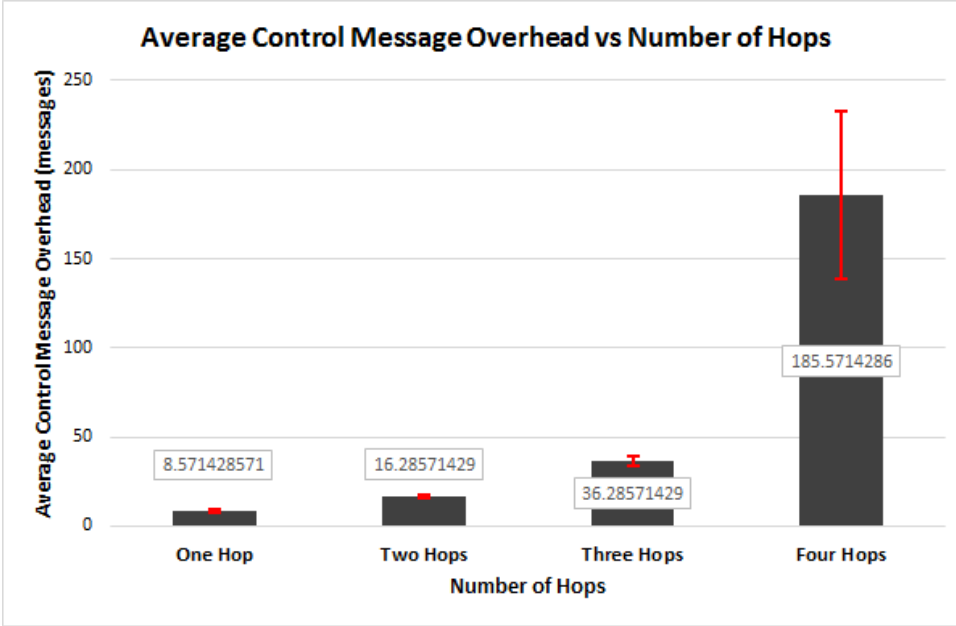


Figure 6.4: Average Control Message Overhead vs Number of Hops

Figure 6.5 shows the distribution of control message overhead versus the hop distance. It reflects the high variation at a large number of hops.

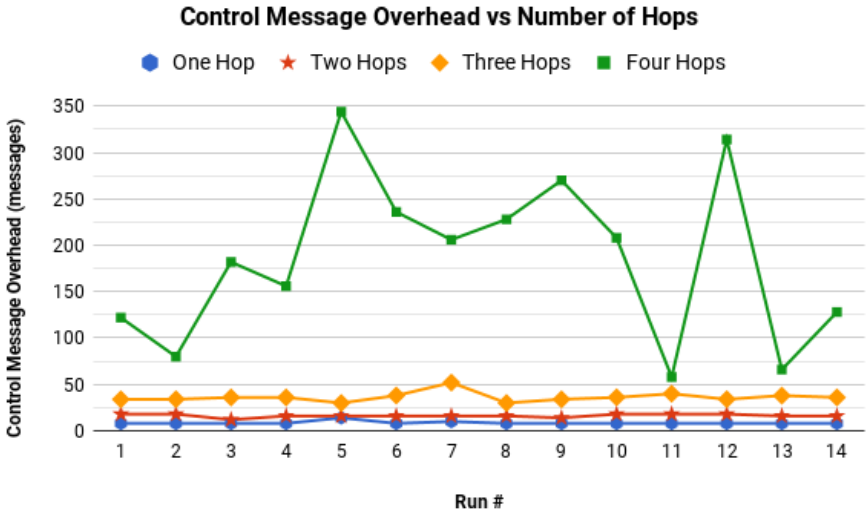


Figure 6.5: Distribution of Control Message Overhead vs Number of Hops

6.3 Radio Link Instability

One important thing observed during the experiments is that the radio links between sensor nodes are not stable and the topology fluctuates frequently. This has been a big challenge in setting up the topologies we want. The level of topology fluctuation increases when the paths in a given topology have many links, as is the case in multi-hop topologies. This instability becomes a challenge for the SDN controller to have a clear view of the topology and compute the shortest path. As a result, the controller usually takes more time in installing flow rules and setting up the right path. More flow entry requests are sent and packets are dropped until the routes are set by installing flow rules. Therefore, the unstable radio links affect the performance measurements and they contribute to the big variation in packet loss and control message overhead values at larger hop distances, as shown in Figures 5.7 and 6.5.

6.4 Effect of Packet Generation Intensity

Two scenarios were used to examine the effect of packet generation and arrival intensity on some of the performance parameters. Packets were generated at 5 second and 1 second interval. As shown in Figures 6.6 and 6.7, the results obtained show

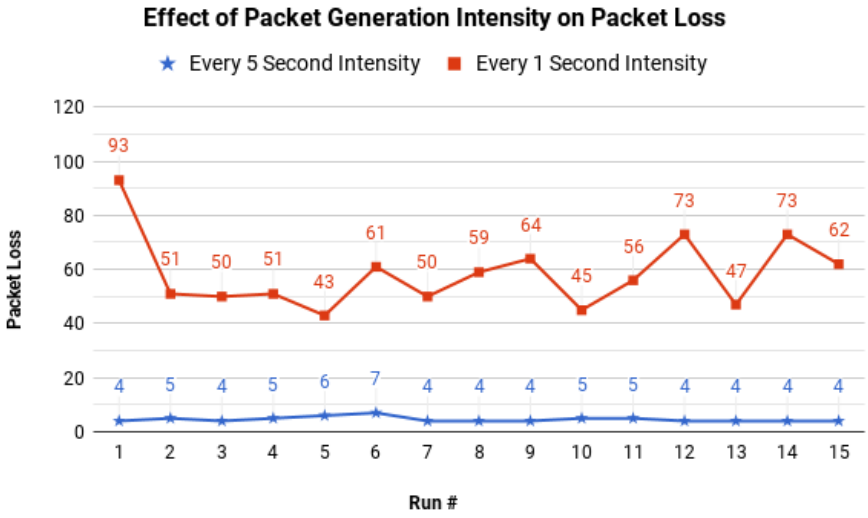


Figure 6.6: Effect of Packet Generation Intensity on Packet Loss

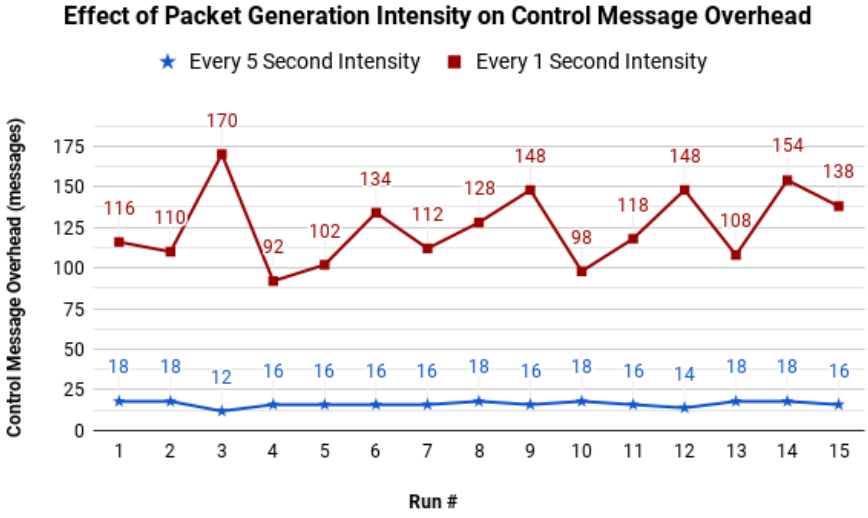


Figure 6.7: Effect of Packet Generation Intensity

that as the rate of packet generation and arrival increases, the packet loss and control message overhead also increases. This can be contributed to the fact that it takes some time to establish paths in the network and the more packets arrive within this

time span, the more packet loss and requests for flow entries occur.

6.5 Robustness: Mobility Handling

Robustness and flexibility are where SDN capitalizes. With its programmable nature, SDN-based WSN can be programmed to handle the dynamicity and mobility that is usually inherent to WSN. This makes SDN-based WSN flexible and more manageable. The centralized intelligence of the SDN controller also makes decision making and management of WSN easier. This was witnessed when we were able to configure the flow table of each sensor node from one central station, and when an SDN controller was developed that detects mobility and makes decisions accordingly. The APIs provided by SDN-WISE were used for the applications developed in the sensor nodes. In addition to that, different python libraries were used in developing the SDN controller.

6.6 Thesis Limitations

Even though the research has achieved its objectives, combining both implementation and experimentation, it also had some limitations. The following are some of the limitations:

- WSN usually consists of many sensor nodes. However, the testbed we used consisted of only 7 sensor nodes for the sake of simplicity and ease of management. It would have been better if a testbed with more sensor nodes is used. However, since the methodology we used is actual hardware-based measurement, it was difficult to manage and carry out measurements when there are many nodes. Simulation tools can be helpful in managing large number of nodes, though simulation can not depict reality and has its own limitations.
- Normally, measurements have to be carried out many times to get more accurate average results. However, doing experiment and measurement in real hardware was very tiresome and took much time. As a result, the sample size we used to average values is small.
- Synchronization is very important when measuring delay and other performance parameters. Internet based synchronization was used in our case, and it was not possible to achieve absolute synchronization.
- The experimental procedures like resetting, removing and logging the output of the sensor nodes were carried out manually. If time allowed, automated ways of doing it could have minimized the time spent on the experiments.

Chapter 7
Conclusion

The overall objective of this thesis work has been to study if SDN technology can be implemented in a network of small wireless-capable and resource-constrained devices. This was successfully demonstrated by developing a real hardware-based testbed. Moreover, performance evaluation of the testbed was carried out. The following points are drawn as a conclusion from the course of this thesis work:

- SDN is still at an early stage regarding its use in a network of small wireless-capable and resource-constrained devices. However, there are still ongoing efforts like that of SDN-WISE. Even though SDN-WISE has the potential to be an SDN solution for a network of resource-constrained devices, it is not well documented and standardized, lacking also an active developer community. Further implementation, standardization, and optimization of the SDN-WISE codebase has to be carried out to make it a complete solution. It has some serious limitations that we have tried to solve in this thesis work.

- The experimental works we did have shown us how flexible SDN is in terms of handling mobility and dynamic changes in the network. This was achieved by improving the SDN-WISE and developing our own solutions like a controller. We were able to install flow entries and clear flow table of each sensor node in the network from one centralized station. This gives great control over the network. Robustness, flexibility, and ease of management are some of the great qualities that all networks and WSN, in particular, should possess. SDN has the potential to provide these qualities.

- In SDN, flow rules can be installed in two ways: proactive and reactive ways. In our testbed, the common reactive approach was used as it was not convenient to carry out the experiments by manually installing flow rules in each node. In reactive approach, the sensor nodes make flow requests to the controller, and then the controller responds and installs flow rules to set up a path for routing data packets. However, these flow request and response messages (control

messages), and the associated packet loss, can be reduced by pre-installing flow entries proactively. If the flow rules are pre-installed, the sensor nodes no more make flow requests. This helps to minimize the control message overhead and thereby reserving the bandwidth of the network for some other important tasks like reporting data. Therefore, performance of SDN can be improved by using a proactive approach of installing flow rules, instead of waiting for the sensor nodes to make a request for flow entries reactively.

- This thesis work has practically shown that SDN can be deployed in a network of small wireless-capable and resource-constrained devices. The performance evaluations have been presented. A similar research on performance analysis of WSNs using RPL as routing protocol, in a COOJA simulated environment, was carried out by Ali [Ali12]. They obtained an average convergence time of around 19 seconds. Whereas, the results we obtained for convergence time range from 19.95 to 45s. This shows that the performance of SDN is adequate, especially taking into account that our results include a realistic environment with variable link conditions. Therefore, we recommend for the adoption of SDN in a network of small wireless-capable and resource-constrained devices.

- A correlation observed in the experiments is the effect of topology, hop distance, rate of packet arrival, and radio link unstability on the performance of the testbed. It was observed that as the topology grows with more sensor nodes and becomes more complex, the performance declines. The same is true when the hop distance and rate of packet arrivals increase also. The unstable radio links were also contributing to the slow convergence time and more control message overheads which negatively affect the performance.

- The testbed developed in this thesis can be used as a benchmark for future studies. Optimization of the implementation and standardization of communication protocols can be further studied. We can also see the possibility for implementation of energy aware routing algorithms in the SDN controller that makes routing decisions taking into account the battery levels of each sensor node.

References

[20114] Terminology for Constrained-Node Networks. https://tools.ietf.org/pdf/rfc7228.pdf, 2014. Accessed: 2017-02-25.

[Ali12] Hazrat Ali. A Performance Evaluation of RPL in Contiki. *Master Thesis*, pages 1–91, 2012.

[AMN+14] Bruno Nunes Astuto, Marc Mendonca, Xuan Nam Nguyen, Katia Obraczka, and Thierry Turletti. A Survey of Software-Defined Networking : Past , Present , and Future of Programmable Networks To cite this version : A Survey of Software-Defined Networking : Past , Present , and Future of Programmable Networks. *IEEE Communications Surveys & Tutorials*, 16:1617–1634, 2014.

[ASS+14] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, Erdal Cayirci, Bruno Astuto A Nunes, Marc Mendonca, Xuan-nam Nguyen, Katia Obraczka, Thierry Turletti, Open Access, Athanassios Boulis, Chih-chieh Han, Mani B Srivastava, Gurwinder Kaur, Rachit Mohan Garg, Luca Mottola, and Gian Pietro Picco. Programming Wireless Sensor Networks : Fundamental Concepts and State of the Art. 3(3):1–17, 2014.

[ASSC02] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. ACCEPTED FROM OPEN CALL A Survey on Sensor Networks. (August):102–114, 2002.

[BK16] Nikos Bizanis and Fernando Kuipers. SDN and virtualization solutions for the Internet of Things: A survey. *IEEE Access*, PP(99):5591–5606, 2016.

[BMRO16] Samaresh Bera, Sudip Misra, Sanku Kumar Roy, and Mohammad S. Obaidat. Soft-WSN: Software-Defined WSN Management System for IoT Applications. *IEEE Systems Journal*, pages 1–8, 2016.

[CGMP12] Salvatore Costanzo, Laura Galluccio, Giacomo Morabito, and Sergio Palazzo. Software defined wireless networks: Unbridling SDNs. *Proceedings - European Workshop on Software Defined Networks, EWSDN 2012*, pages 1–6, 2012.

[DGAM14] Alejandro De Gante, Mohamed Aslan, and Ashraf Matrawy. Smart wireless sensor network management based on software-defined networking. *2014 27th Biennial Symposium on Communications (QBSC)*, pages 71–75, 2014.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. *Proceedings - Conference on Local Computer Networks, LCN*, pages 455–462, 2004.

[DKB11] Peter Dely, Andreas Kassler, and Nico Bayer. OpenFlow for Wireless Mesh Networks. *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6, 2011.

[FGN15] Olivier Flauzac, Carlos Gonzalez, and Florent Nolot. SDN Based Architecture for Clustered WSN. *Proceedings - 2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2015*, pages 342–347, 2015.

[FS17] Roy Friedman and David Sainz. An Architecture for SDN Based Sensor Networks. 2017.

[GKP+08] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[GMMPa] Laura Galluccio, Sebastiano Milardo, Giacomo Morabito, and Sergio Palazzo. SDN-WISE - The stateful Software Defined Networking solution for the Internet of Things. http://sdn-wise.dieei.unict.it/. Accessed: 2017-02-25.

[GMMPb] Laura Galluccio, Sebastiano Milardo, Giacomo Morabito, and Sergio Palazzo. SDN-WISE Core Part 1. http://sdn-wise.dieei.unict.it/docs/guides/CorePart1.html. Accessed: 2017-02-25.

[GMMP15a] Laura Galluccio, Sebastiano Milardo, Giacomo Morabito, and Sergio Palazzo. A "wise" choice for Wireless Sensor Networks Management, Experimentaion, and Applicaion Development. http://sdn-wise.dieei.unict.it/docs/slides/sdn_wise_cavalese.pdf, 2015.

[GMMP15b] Laura Galluccio, Sebastiano Milardo, Giacomo Morabito, and Sergio Palazzo. Reprogramming Wireless Sensor Networks by using SDN-WISE: A hands-on demo. *Proceedings - IEEE INFOCOM*, 2015-Augus:19–20, 2015.

[GMMP15c] Laura Galluccio, Sebastiano Milardo, Giacomo Morabito, and Sergio Palazzo. SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks. *Proceedings - IEEE INFOCOM*, 26:513–521, 2015.

[HR14] Zhi-jie Han and Wanli Ren. A Novel Wireless Sensor Networks Structure Based on the SDN. *International Journal of Distributed Sensor Networks*, 2014(7):1–7, 2014.

[Int] Internet Engineering Task Force (IETF). RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. https://tools.ietf.org/html/rfc6550. Accessed: 2017-04-15.

[KRV+14] Diego Kreutz, Fernando M. V. Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. pages 1–61, 2014.

[LAB+16] Antonio Liñán, Colina Alvaro, Vives Antoine Bagula, Marco Zennaro, and Ermanno Pietrosemoli. *Internet of Things in 5 Days*. 2016.

[LTQ12] Tie Luo, Hwee Pink Tan, and Tony Q S Quek. Sensor openflow: Enabling software-defined wireless sensor networks. *IEEE Communications Letters*, 16(11):1896–1899, 2012.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69, 2008.

[MR11] Arif Mahmud and Rahim Rahmani. Exploitation of OpenFlow in wireless sensor networks. *Proceedings of 2011 International Conference on Computer Science and Network Technology, ICCSNT 2011*, 1:594–600, 2011.

[Msp] The MSP430 UART. http://www.simplyembedded.org/tutorials/msp430-uart/. Accessed: 2017-05-21.

[O.N12] O.N.F. Software-defined networking: The new norm for networks. *ONF White Paper*, 2:2–6, 2012.

[Opea] Open Network Foundation. ONF Overview. https://www.opennetworking.org/about/onf-overview. Accessed: 2017-04-15.

[Opeb] Open Network Foundation. Software-Defined Networking (SDN) Definition. https://www.opennetworking.org/sdn-resources/sdn-definition. Accessed: 2017-04-15.

[SGY+09] Rob Sherwood, Glen Gibb, Kok-kiong Yap, Guido Appenzeller, Martin Casado, Nick Mckeown, and Guru Parulkar. FlowVisor: A Network Virtualization Layer. *Network*, page 15, 2009.

[SSP16] Kshira Sagar Sahoo, Bibhudatta Sahoo, and Abinas Panda. A secured SDN framework for IoT. *Proceedings - 2015 International Conference on Man and Machine Interfacing, MAMI 2015*, (DECEMBER 2015), 2016.

[Yan12] Yitian Yan. Wireless Sensor Networks - Cooperation and Network Coding for Performance Enhancement: Theories and Experiments. pages 1–58, 2012.

[Zol10] Zolertia. Z1 Datasheet. http://zolertia.sourceforge.net/wiki/images/e/e8/Z1_RevC_Datasheet.pdf, 2010. Accessed: 2017-04-15.

Appendix A
Selected Source Code

The source code of our testbed implementation can be accessed from github at this url: https://github.com/miarcompanies/sdn-wise-contiki. However, the most important contributions while implementing the testbed are attached in this Appendix.

A.1 PythonController.py - Our Own Python-Based Simple SDN Controller

```
#!/usr/bin/env python
import sys
import time
import serial
import networkx as nx
import threading
from threading import Thread
from datetime import datetime
import networkx as nx
def readUART(Topo):
    try:
        ser = serial.Serial('/dev/ttyUSB0',115200)
        prev_length = []
        length = []
        for t in range(10):
            prev_length.append(0)
            length.append(0)
        while 1:
            mtype = ser.readline()
            if 'Report' in mtype:
                topo = ser.readline()
                print 'Topo:'+topo
                topoarray = map(int, topo.split(","))
                print datetime.now().strftime('%H:%M:%S.%f')
                print "Topo in Array:"
```

```
            print topoarray
            for s in range(10): #10 nodes assumed
                if topoarray[0] == s+1:
                    length[s] = len(topoarray)
                    print length[s]
                    if length[s] != prev_length[s]:#topology changes
                        Topo.clear()
                    prev_length[s] = length[s]
            Topo.add_node(topoarray[0])
            for num in range(2,len(topoarray)-2,3):
                Topo.add_node(topoarray[num])
                Topo.add_edge(topoarray[0], topoarray[num],weight=topoarray
                    [num+2])
                Topo.add_edge(topoarray[num], topoarray[0],weight=topoarray
                    [num+2])
            print Topo.nodes()
            print Topo.edges(data=True)
        elif 'Request' in mtype:
            req = ser.readline()
            print datetime.now().strftime('%H:%M:%S.%f')
            print 'Request:'+req
            reqarray = map(int, req.split(","))
            print "Request in Array:"
            print reqarray
            print 'Shortest Path from %d to %d: ' % (reqarray[0], reqarray
                [2])
            try:
                shortpath = nx.dijkstra_path(Topo,reqarray[0],reqarray[2],
                    weight=True)
                print shortpath
                if (len(shortpath) > 2):
                    nxh = shortpath[1]
                    for x in range(len(shortpath)-1):
                        unicastcommand = str(shortpath[0]-1)
                        unicastcommand += str(shortpath[0]-1)
                        unicastcommand += 'u'
                        unicastcommand += str(shortpath[x+1]-1)
                        unicastcommand += str(nxh-1)
                        unicastcommand += str('\n')
                        print "Unicast Command to send: "+unicastcommand
                        bauni = bytearray(unicastcommand)
                        ser.write(bauni)
                        print datetime.now().strftime('%H:%M:%S.%f')
                        print "Command written to serial port"
                else:
```

```
                    unicastcommand = str(shortpath[0]-1)
                    unicastcommand += str(shortpath[0]-1)
                    unicastcommand += 'u'
                    unicastcommand += str(shortpath[1]-1)
                    unicastcommand += str(shortpath[1]-1)
                    unicastcommand += str('\n')
                    print "Unicast Command to send: "+unicastcommand
                    bauni = bytearray(unicastcommand)
                    ser.write(bauni)
                    print datetime.now().strftime('%H:%M:%S.%f')
                    print "Command written to serial port"
    except Exception:
           print "Node %d not reachable from %d" % (reqarray[2],reqarray[0])
            else:
                print mtype
    except (KeyboardInterrupt):
        sys.exit()
def writeUART(Topo):
    try:
        ser = serial.Serial('/dev/ttyUSB0',115200)
        print 'Please enter your command - write Exit to quit\n'
        status = sys.stdin.readline()
        while 1:
            ba = bytearray(status)
            ser.write(ba)
            if status == 'Exit':
                ser.close()
                sys.exit()
                break
            print 'Please enter your command - write Exit to quit\n'
            status = sys.stdin.readline()
    except (KeyboardInterrupt):
        sys.exit()
if __name__=='__main__':
    print("Simple Python Controller for SDN-WISE Starting .....")
    print datetime.now().strftime('%H:%M:%S.%f')
    Topo = nx.DiGraph()
    threadwrite = threading.Thread(target = writeUART, args = [Topo])
    threadwrite.Daemon = True
    threadwrite.start()
    threadread = threading.Thread(target = readUART, args = [Topo])
    threadread.Daemon = True
    threadread.start()
```

A.2 Extensions to the SDN-WISE Codebase

A.2.1 sdnwise.c

```
#include "contiki.h"
#include "net/rime/rime.h"
#include "net/linkaddr.h"
#include "dev/watchdog.h"
#include "dev/uart0.h"
#include "dev/leds.h"
#include "lib/list.h"
#if CFS_ENABLED
#include "cfs/cfs.h"
#endif
#if ELF_ENABLED
#include "loader/elfloader.h"
#endif
#include "sdn-wise.h"
#include "flowtable.h"
#include "packet-buffer.h"
#include "packet-handler.h"
#include "packet-creator.h"
#include "neighbor-table.h"
#include "node-conf.h"
#define UART_BUFFER_SIZE   MAX_PACKET_LENGTH
#define UNICAST_CONNECTION_NUMBER 29
#define BROADCAST_CONNECTION_NUMBER   30
#define TIMER_EVENT     50
#define UPDATE_TOPO_EVENT  60
#define RF_U_SEND_EVENT  51
#define RF_B_SEND_EVENT  52
#define RF_U_RECEIVE_EVENT 53
#define RF_B_RECEIVE_EVENT 54
#define UART_RECEIVE_EVENT   55
#define RF_SEND_BEACON_EVENT 56
#define RF_SEND_REPORT_EVENT 57
#define NEW_PACKET_EVENT 58
#define ACTIVATE_EVENT 59
#define DEBUG 1
#if DEBUG && (!SINK || DEBUG_SINK)
#include <stdio.h>
#include <inttypes.h>
#define PRINTF(...) printf(__VA_ARGS__)
#else
#define PRINTF(...)
#endif
```

```
#define STR(x) #x
#define SHOW_DEFINE(x) printf("%s=%s\n", #x, STR(x))
PROCESS(main_proc, "Main Process");
PROCESS(rf_u_send_proc, "RF Unicast Send Process");
PROCESS(rf_b_send_proc, "RF Broadcast Send Process");
PROCESS(packet_handler_proc, "Packet Handler Process");
PROCESS(timer_proc, "Timer Process");
PROCESS(beacon_timer_proc, "Beacon Timer Process");
PROCESS(report_timer_proc, "Report Timer Process");
AUTOSTART_PROCESSES(&main_proc, &rf_u_send_proc, &rf_b_send_proc, &timer_proc
    , &beacon_timer_proc, &report_timer_proc, &packet_handler_proc);
static uint8_t uart_buffer[UART_BUFFER_SIZE];
static uint8_t uart_buffer_index = 0;
static uint8_t uart_buffer_expected = 0;
static uint8_t tmp_uart_buffer[5];
static uint8_t copy_to_tmp = 0;
static uint8_t tmp_index = 0;
uint8_t data_packet_counter = 0;
void rf_unicast_send(packet_t* p)
{
   process_post(&rf_u_send_proc, RF_U_SEND_EVENT, (process_data_t)p);
}
void rf_broadcast_send(packet_t* p)
{
   process_post(&rf_b_send_proc, RF_B_SEND_EVENT, (process_data_t)p);
}
static void unicast_rx_callback(struct unicast_conn *c, const linkaddr_t *
    from)
{
   packet_t* p = get_packet_from_array((uint8_t *)packetbuf_dataptr());
   if (p != NULL){
     p->info.rssi = (uint8_t) (- packetbuf_attr(PACKETBUF_ATTR_RSSI));
     process_post(&main_proc, RF_U_RECEIVE_EVENT, (process_data_t)p);
   }
}
static void broadcast_rx_callback(struct broadcast_conn *c, const linkaddr_t
    *from)
{
   packet_t* p = get_packet_from_array((uint8_t *)packetbuf_dataptr());
   if (p != NULL){
     p->info.rssi = (uint8_t) (- packetbuf_attr(PACKETBUF_ATTR_RSSI));
     process_post(&main_proc, RF_B_RECEIVE_EVENT, (process_data_t)p);
   }
}
int uart_rx_callback(unsigned char c)
```

```
{
    if(uart_buffer_index == UART_BUFFER_SIZE)
     uart_buffer_index = 0;
    uart_buffer[uart_buffer_index] = c;
    if(uart_buffer_index<5)
     tmp_uart_buffer[uart_buffer_index] = uart_buffer[uart_buffer_index];
       if(copy_to_tmp == 1 && tmp_index<5){
           tmp_uart_buffer[tmp_index] = uart_buffer[uart_buffer_index];
           tmp_index ++;
       }
    if(c == 10){
       copy_to_tmp = 1;
       tmp_index = 0;
    }
    uart_buffer_index++;
    if((tmp_index == 5 || uart_buffer_index == 5) && ((tmp_uart_buffer[0] ==
        100) || (tmp_uart_buffer[2] == 117 || tmp_uart_buffer[2] == 98 ||
        tmp_uart_buffer[2] == 100) || (tmp_uart_buffer[2]== 114 &&
        tmp_uart_buffer[3] == 102) || (tmp_uart_buffer[2]== 115 &&
        tmp_uart_buffer[3] == 102) || (tmp_uart_buffer[2] == 116 &&
        tmp_uart_buffer[4] == 114) )){
       copy_to_tmp = 0;
       tmp_index = 0;
       packet_t* p = create_packet_empty();
       p->header.net = conf.my_net;
       if(tmp_uart_buffer[0] == 49 && tmp_uart_buffer[1] == 49){ //'1'
          p->header.dst.u8[0] = 2;
          p->header.dst.u8[1] = 0;
       }
       else if(tmp_uart_buffer[0] == 50 && tmp_uart_buffer[1] == 50){//'2'
           p->header.dst.u8[0] = 3;
           p->header.dst.u8[1] = 0;
       }
       else if(tmp_uart_buffer[0] == 51 && tmp_uart_buffer[1] == 51){ //'3'
           p->header.dst.u8[0] = 4;
           p->header.dst.u8[1] = 0;
       }
       else if(tmp_uart_buffer[0] == 52 && tmp_uart_buffer[1] == 52){//'4'
           p->header.dst.u8[0] = 5;
           p->header.dst.u8[1] = 0;
       }
       else if(tmp_uart_buffer[0] == 53 && tmp_uart_buffer[1] == 53){//'5'
           p->header.dst.u8[0] = 6;
           p->header.dst.u8[1] = 0;
       }
```

```
else if(tmp_uart_buffer[0] == 54 && tmp_uart_buffer[1] == 54){//'6'
    p->header.dst.u8[0] = 7;
    p->header.dst.u8[1] = 0;
}
else if(tmp_uart_buffer[0] == 55 && tmp_uart_buffer[1] == 55){//'7'
    p->header.dst.u8[0] = 8;
    p->header.dst.u8[1] = 0;
}
else if(tmp_uart_buffer[0] == 56 && tmp_uart_buffer[1] == 56){//'8'
    p->header.dst.u8[0] = 9;
    p->header.dst.u8[1] = 0;
}
else if(tmp_uart_buffer[0] == 57 && tmp_uart_buffer[1] == 57){//'9'
    p->header.dst.u8[0] = 10;
    p->header.dst.u8[1] = 0;
}
else{
    p->header.dst.u8[0] = 1;
    p->header.dst.u8[1] = 0;
}
p->header.src = conf.my_address;
if(tmp_uart_buffer[0] == 100){ //dd for data d in ascii is 100
    p->header.typ = DATA;
    if(tmp_uart_buffer[3] == 48 && tmp_uart_buffer[4] == 48){ //'00'
        p->header.dst.u8[0] = 1;
     p->header.dst.u8[1] = 0;
    }
    else if(tmp_uart_buffer[3] == 49 && tmp_uart_buffer[4] == 49){ //
        '11'
        p->header.dst.u8[0] = 2;
        p->header.dst.u8[1] = 0;
    }
    else if(tmp_uart_buffer[3] == 50 && tmp_uart_buffer[4] == 50){//
        '2'
        p->header.dst.u8[0] = 3;
        p->header.dst.u8[1] = 0;
    }
    else if(tmp_uart_buffer[3] == 51 && tmp_uart_buffer[4] == 51){ //
        '3'
      p->header.dst.u8[0] = 4;
      p->header.dst.u8[1] = 0;
    }
    else if(tmp_uart_buffer[3] == 52 && tmp_uart_buffer[4] == 52){//
        '4'
      p->header.dst.u8[0] = 5;
```

```
      p->header.dst.u8[1] = 0;
    }
    else if(tmp_uart_buffer[3] == 53 && tmp_uart_buffer[4] == 53){//
        '5'
      p->header.dst.u8[0] = 6;
      p->header.dst.u8[1] = 0;
    }
    else if(tmp_uart_buffer[3] == 54 && tmp_uart_buffer[4] == 54){//
        '6'
       p->header.dst.u8[0] = 7;
       p->header.dst.u8[1] = 0;
    }
    else if(tmp_uart_buffer[3] == 55 && tmp_uart_buffer[4] == 55){//
        '7'
       p->header.dst.u8[0] = 8;
       p->header.dst.u8[1] = 0;
    }
    else if(tmp_uart_buffer[3] == 56 && tmp_uart_buffer[4] == 56){//
        '8'
       p->header.dst.u8[0] = 9;
       p->header.dst.u8[1] = 0;
    }
    else if(tmp_uart_buffer[3] == 57 && tmp_uart_buffer[4] == 57){//
        '9'
       p->header.dst.u8[0] = 10;
       p->header.dst.u8[1] = 0;
    }
    else{
       p->header.dst.u8[0] = 1;
       p->header.dst.u8[1] = 0;
   }
   if(tmp_uart_buffer[1] == 48 && tmp_uart_buffer[2] == 48){//'00'
      p->header.src.u8[0] = 1;
      p->header.src.u8[1] = 0;
   }
   else if(tmp_uart_buffer[1] == 49 && tmp_uart_buffer[2] == 49){//
       '11'
      p->header.src.u8[0] = 2;
      p->header.src.u8[1] = 0;
   }
   else if(tmp_uart_buffer[1] == 50 && tmp_uart_buffer[2] == 50){//
       '22'
      p->header.src.u8[0] = 3;
      p->header.src.u8[1] = 0;
    }
```

```
    else if(tmp_uart_buffer[1] == 51 && tmp_uart_buffer[2] == 51){//
        '33'
         p->header.src.u8[0] = 4;
         p->header.src.u8[1] = 0;
    }
    else if(tmp_uart_buffer[1] == 52 && tmp_uart_buffer[2] == 52){//
        '44'
         p->header.src.u8[0] = 5;
         p->header.src.u8[1] = 0;
    }
    else if(tmp_uart_buffer[1] == 53 && tmp_uart_buffer[2] == 53){//
        '55'
         p->header.src.u8[0] = 6;
         p->header.src.u8[1] = 0;
    }
    else if(tmp_uart_buffer[1] == 54 && tmp_uart_buffer[2] == 54){//
        '66'
         p->header.src.u8[0] = 7;
         p->header.src.u8[1] = 0;
    }
    else if(tmp_uart_buffer[1] == 55 && tmp_uart_buffer[2] == 56){//
        '77'
         p->header.src.u8[0] = 8;
         p->header.src.u8[1] = 0;
    }
    else if(tmp_uart_buffer[1] == 56 && tmp_uart_buffer[2] == 57){//
        '88'
         p->header.src.u8[0] = 9;
         p->header.src.u8[1] = 0;
    }
    else if(tmp_uart_buffer[1] == 57 && tmp_uart_buffer[2] == 57){//
        '99'
         p->header.src.u8[0] = 10;
         p->header.src.u8[1] = 0;
    }
}
else{
    p->header.typ = CONFIG;
}
p->header.nxh = conf.my_address;;
set_payload_at(p, 0, tmp_uart_buffer[0]);
set_payload_at(p, 1, tmp_uart_buffer[1]);
set_payload_at(p, 2, tmp_uart_buffer[2]);
set_payload_at(p, 3, tmp_uart_buffer[3]);
set_payload_at(p, 4, tmp_uart_buffer[4]);
```

```
        if (p != NULL){
        p->info.rssi = 255;
        if(p->header.typ == DATA){
         set_payload_at(p, 5, data_packet_counter);
         print_report_data(tmp_uart_buffer[1], tmp_uart_buffer[2],
             tmp_uart_buffer[3], tmp_uart_buffer[4]);
         PRINTF("Send Data Packet %d\n", data_packet_counter);
         data_packet_counter++;
        }
        else{
         print_report_config(tmp_uart_buffer[0], tmp_uart_buffer[1],
             tmp_uart_buffer[3], tmp_uart_buffer[4]);
        }
        process_post(&main_proc, UART_RECEIVE_EVENT, (process_data_t)p);
      }
    }
    return 0;
}
static const struct unicast_callbacks unicast_callbacks = {
unicast_rx_callback
};
static struct unicast_conn uc;
static const struct broadcast_callbacks broadcast_callbacks = {
    broadcast_rx_callback
};
static struct broadcast_conn bc;
PROCESS_THREAD(main_proc, ev, data) {
    PROCESS_BEGIN();
    uart0_init(BAUD2UBR(115200));    /* set the baud rate as necessary */
    uart0_set_input(uart_rx_callback); /* set the callback function */

    node_conf_init();
    flowtable_init();
    packet_buffer_init();
    neighbor_table_init();
    address_list_init();
    leds_init();
    #if SINK
    print_packet_uart(create_reg_proxy());
    #endif
    while(1) {
      PROCESS_WAIT_EVENT();
      switch(ev) {
        case TIMER_EVENT:
            PRINTF("Neighbor Table\n");
```

```
    print_neighbor_table();
    reset_isalive_neighbor();
    PRINTF("Neighbor Table\n");
    print_neighbor_table();
    break;
case UPDATE_TOPO_EVENT:
    PRINTF("Updating Topology Neighbors\n");
    PRINTF("Neighbor Table\n");
    print_neighbor_table();
    update_topo_neighbor();
    PRINTF("Neighbor Table\n");
    print_neighbor_table();
    break;
case UART_RECEIVE_EVENT:
    leds_toggle(LEDS_GREEN);
    process_post(&packet_handler_proc, NEW_PACKET_EVENT, (
        process_data_t)data);
    break;
case RF_B_RECEIVE_EVENT:
    leds_toggle(LEDS_YELLOW);
    if (!conf.is_active){
      conf.is_active = 1;
      process_post(&beacon_timer_proc, ACTIVATE_EVENT, (process_data_t
          )NULL);
      process_post(&report_timer_proc, ACTIVATE_EVENT, (process_data_t
          )NULL);
    }
case RF_U_RECEIVE_EVENT:
    process_post(&packet_handler_proc, NEW_PACKET_EVENT, (
        process_data_t)data);
    break;
case RF_SEND_BEACON_EVENT:
    leds_toggle(LEDS_RED);
PRINTF("Beacon Send ");
    rf_broadcast_send(create_beacon());
    break;
case RF_SEND_REPORT_EVENT:
    leds_toggle(LEDS_RED);
    PRINTF("Send Report\n");
    #if !SINK
    rf_unicast_send(create_report());
     #else
    PRINTF("SINK Sending Report - To Controller(Method will be
        developed)\n");
    send_report_to_controller(create_report());
```

```
            #endif
            break;
        }
    }
    PROCESS_END();
}
PROCESS_THREAD(rf_u_send_proc, ev, data) {
static linkaddr_t addr;
PROCESS_EXITHANDLER(unicast_close(&uc);)
PROCESS_BEGIN();
unicast_open(&uc, UNICAST_CONNECTION_NUMBER, &unicast_callbacks);
while(1) {
  PROCESS_WAIT_EVENT_UNTIL(ev == RF_U_SEND_EVENT);
  packet_t* p = (packet_t*)data;
  if (p != NULL){
    p->header.ttl--;
    if (!is_my_address(&(p->header.dst))){
      int i = 0;
      int sent_size = 0;
      if (LINKADDR_SIZE < ADDRESS_LENGTH){
        sent_size = LINKADDR_SIZE;
      } else {
        sent_size = ADDRESS_LENGTH;
      }
      for ( i=0; i<sent_size; ++i){
        addr.u8[i] = p->header.nxh.u8[(sent_size-1)-i];
      }
      addr.u8[0] = p->header.nxh.u8[0];
      addr.u8[1] = p->header.nxh.u8[1];
      PRINTF("[TXU]: ");
      print_packet(p);
      uint8_t* a = (uint8_t*)p;
      packetbuf_copyfrom(a,p->header.len);
      unicast_send(&uc, &addr);
    }
#if SINK
    else {
     if(p->header.typ == REPORT)
            PRINTF("SINK Sending Report Packet - To Controller(Method will be
                 developed)\n");
 send_report_to_controller(p);
}
#endif
    packet_deallocate(p);
  }
```

```
}
PROCESS_END();
}
PROCESS_THREAD(rf_b_send_proc, ev, data) {
PROCESS_EXITHANDLER(broadcast_close(&bc);)
PROCESS_BEGIN();
broadcast_open(&bc, BROADCAST_CONNECTION_NUMBER, &broadcast_callbacks);
while(1) {
  PROCESS_WAIT_EVENT_UNTIL(ev == RF_B_SEND_EVENT);
  packet_t* p = (packet_t*)data;
  if (p != NULL){
    p->header.ttl--;
    PRINTF("[TXB]: ");
    print_packet(p);
    PRINTF("\n");
    uint8_t* a = (uint8_t*)p;
    packetbuf_copyfrom(a,p->header.len);
    broadcast_send(&bc);
    packet_deallocate(p);
  }
}
PROCESS_END();
}
PROCESS_THREAD(timer_proc, ev, data) {
static struct etimer et;
static struct etimer et_update;
PROCESS_BEGIN();
while(1) {
  etimer_set(&et, 15 * CLOCK_SECOND);
  PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
  etimer_reset(&et);
  process_post(&main_proc, TIMER_EVENT, (process_data_t)NULL);
  etimer_set(&et_update, 10 * CLOCK_SECOND);
  PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et_update));
  etimer_reset(&et_update);
  process_post(&main_proc, UPDATE_TOPO_EVENT, (process_data_t)NULL);
}
PROCESS_END();
}
PROCESS_THREAD(beacon_timer_proc, ev, data) {
static struct etimer et;
PROCESS_BEGIN();
while(1){
#if !SINK
  if (!conf.is_active){
```

```
    PROCESS_WAIT_EVENT_UNTIL(ev == ACTIVATE_EVENT);
  }
#endif
  etimer_set(&et, conf.beacon_period * CLOCK_SECOND);
  PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
  process_post(&main_proc, RF_SEND_BEACON_EVENT, (process_data_t)NULL);
}
PROCESS_END();
}
PROCESS_THREAD(report_timer_proc, ev, data) {
static struct etimer et;
PROCESS_BEGIN();
while(1){
#if !SINK
  if (!conf.is_active){
    PROCESS_WAIT_EVENT_UNTIL(ev == ACTIVATE_EVENT);
  }
#endif
  etimer_set(&et, conf.report_period * CLOCK_SECOND);
  PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
  process_post(&main_proc, RF_SEND_REPORT_EVENT, (process_data_t)NULL);
}
PROCESS_END();
}
PROCESS_THREAD(packet_handler_proc, ev, data) {
PROCESS_BEGIN();
while(1) {
  PROCESS_WAIT_EVENT_UNTIL(ev == NEW_PACKET_EVENT);
  packet_t* p = (packet_t*)data;
  PRINTF("[RX]: ");
  print_packet(p);
  PRINTF("\n");
  handle_packet(p);
}
PROCESS_END();
}
```

A.2.2 neighbor-table.c

```
#include <string.h>
#include "lib/memb.h"
#include "lib/list.h"
#include "packet-buffer.h"
#include "neighbor-table.h"
#include "packet-creator.h"
#include "node-conf.h"
```

```
#define _PACKET_TTL 100;
#define DEBUG 1
#if DEBUG && !SINK
#include <stdio.h>
#define PRINTF(...) printf(__VA_ARGS__)
#else
#define PRINTF(...)
#endif
  LIST(neighbor_table);
  MEMB(neighbors_memb, neighbor_t, (MAX_PAYLOAD_LENGTH) / (NEIGHBOR_LENGTH));
  static neighbor_t * neighbor_allocate(void);
  static void neighbor_free(neighbor_t *);
  static void print_neighbor(neighbor_t*);
  static void
  print_neighbor(neighbor_t* n)
  {
    print_address(&(n->address));
    PRINTF("%d",n->rssi);
    PRINTF(" %d",n->is_alive);
  }
  void print_neighbor_table(void)
  {
    neighbor_t *n;
    for(n = list_head(neighbor_table); n != NULL; n = n->next) {
      PRINTF("[NGT]: ");
      print_neighbor(n);
      PRINTF("\n");
    }
  }
  void purge_neighbor_table(void)
  {
    neighbor_t *n;
    neighbor_t *next;
    for(n = list_head(neighbor_table); n != NULL;) {
      next = n->next;
      neighbor_free(n);
      n = next;
    }
  }
  void reset_isalive_neighbor(void){
    neighbor_t *n;
    neighbor_t *next;

    for(n = list_head(neighbor_table); n != NULL;) {
      next = n;
```

```
      next->is_alive = 0;
      next = n->next;
      n = next;
    }
  }
  void update_topo_neighbor(void){
    neighbor_t *n;
    neighbor_t *next;
    for(n = list_head(neighbor_table); n != NULL;) {
      next = n;
      if(next->is_alive == 0){
if((conf.nxh_vs_sink.u8[0] == next->address.u8[0]) && (conf.nxh_vs_sink.u8
      [1] == next->address.u8[1])){
 conf.nxh_vs_sink.u8[0] = 1;
 conf.nxh_vs_sink.u8[1] = 0;
 conf.hops_from_sink = _PACKET_TTL;
 conf.rssi_from_sink = 100;
}
remove_neighbor(next);
      }
      next = n->next;
      n = next;
    }
  }
  address_t nearest_neighbor(void){
neighbor_t *n;
n = list_head(neighbor_table);
neighbor_t *nearest = n;
for(; n != NULL;n=n->next) {
 if(n->rssi < nearest->rssi)
  nearest = n;
}
return nearest->address;
  }
  static neighbor_t * neighbor_allocate(void)
  {
    neighbor_t *p;
    p = memb_alloc(&neighbors_memb);
    if(p == NULL) {
      PRINTF("[NGT]: Failed to allocate a neighbor\n");
    }
    return p;
  }
  static void neighbor_free(neighbor_t* n)
  {
```

```
  list_remove(neighbor_table, n);
  int res = memb_free(&neighbors_memb, n);
  if (res !=0){
    PRINTF("[NGT]: Failed to free a neighbor. Reference count: %d\n",res);
  }
}
uint8_t neighbor_cmp(neighbor_t* a, neighbor_t* b)
{
  return address_cmp(&(a->address),&(b->address));
}
neighbor_t* neighbor_table_contains(address_t* a)
{
  neighbor_t* tmp;
  for(tmp = list_head(neighbor_table); tmp != NULL; tmp = tmp->next) {
    if(address_cmp(&(tmp->address),a)){
      return tmp;
    }
  }
  return NULL;
}
void add_neighbor(address_t* address, uint8_t rssi, uint8_t is_alive)
{
  neighbor_t* res = neighbor_table_contains(address);
  if (res == NULL){
    neighbor_t* n = neighbor_allocate();
    if (n != NULL){
      memset(n, 0, sizeof(*n));
      n->address = *address;
      n->rssi = rssi;
n->is_alive = is_alive;
      list_add(neighbor_table,n);
    }
  } else {
    res->rssi = rssi;
    res->is_alive = is_alive;
  }
}
void remove_neighbor(neighbor_t* n){
 PRINTF("Removing Neighbor\n");
neighbor_free(n);
}
void fill_payload_with_neighbors(packet_t* p)
{
  uint8_t i = REPORT_INIT_INDEX;
  set_payload_at(p,i,(uint8_t)(list_length(neighbor_table) & 0xFF));
```

```
    i++;
    neighbor_t *n;
    for(n = list_head(neighbor_table); n != NULL; n = n->next) {
      uint8_t j = 0;
      for (j = 0; j < ADDRESS_LENGTH; ++j){
        set_payload_at(p,i,n->address.u8[j]);
        ++i;
      }
      set_payload_at(p,i,n->rssi);
      ++i;
    }
  }
  void neighbor_table_init(void)
  {
    list_init(neighbor_table);
    memb_init(&neighbors_memb);
  }
  void test_neighbor_table(void)
  {
    address_t addr1;
    addr1.u8[0] = 1;
    addr1.u8[1] = 1;
    uint8_t rssi1 = 100;
    address_t addr3;
    addr3.u8[0] = 1;
    addr3.u8[1] = 1;
    uint8_t rssi3 = 50;
    address_t addr2;
    addr2.u8[0] = 2;
    addr2.u8[1] = 2;
    uint8_t rssi2 = 200;
    if (!list_length(neighbor_table)){
      add_neighbor(&addr1,rssi1,1);
      add_neighbor(&addr2,rssi2,1);
      add_neighbor(&addr3,rssi3,1);
      print_neighbor_table();
    }else{
      purge_neighbor_table();
      print_neighbor_table();
    }
  }
```

A.2.3 packet-handler.c

```
#include <string.h>
#include <stdio.h>
```

```
#include "contiki.h"
#include "dev/watchdog.h"
#include "packet-handler.h"
#include "address.h"
#include "packet-buffer.h"
#include "packet-creator.h"
#include "neighbor-table.h"
#include "flowtable.h"
#include "node-conf.h"
#include "sdn-wise.h"
#include "net/rime/rime.h"
typedef enum conf_id{
  RESET,
  MY_NET,
  MY_ADDRESS,
  PACKET_TTL,
  RSSI_MIN,
  BEACON_PERIOD,
  REPORT_PERIOD,
  RULE_TTL,
  ADD_ALIAS,
  REM_ALIAS,
  GET_ALIAS,
  ADD_RULE,
  REM_RULE,
  GET_RULE,
  ADD_FUNCTION,
  REM_FUNCTION,
  GET_FUNCTION
} conf_id_t;
const uint8_t conf_size[RULE_TTL+1] =
{
  0,
  sizeof(conf.my_net),
  sizeof(conf.my_address),
  sizeof(conf.packet_ttl),
  sizeof(conf.rssi_min),
  sizeof(conf.beacon_period),
  sizeof(conf.report_period),
  sizeof(conf.rule_ttl)
};
const void* conf_ptr[RULE_TTL+1] =
{
  NULL,
  &conf.my_net,
```

```
  &conf.my_address,
  &conf.packet_ttl,
  &conf.rssi_min,
  &conf.beacon_period,
  &conf.report_period,
  &conf.rule_ttl,
};

#define CNF_READ 0
#define CNF_WRITE 1

#define DEBUG 1
#if DEBUG && (!SINK || DEBUG_SINK)
#define PRINTF(...) printf(__VA_ARGS__)
#else
#define PRINTF(...)
#endif
static void handle_beacon(packet_t*);
static void handle_data(packet_t*);
static void handle_report(packet_t*);
static void handle_response(packet_t*);
static void handle_open_path(packet_t*);
static void handle_config(packet_t*);
static void handle_request(packet_t*);
void handle_packet(packet_t* p)
{
    if (p->info.rssi >= conf.rssi_min && p->header.net == conf.my_net){
      if (p->header.typ == BEACON){
        PRINTF("[PHD]: Beacon %d from %d.%d\n", get_payload_at(p,2), p->
            header.src.u8[0], p->header.src.u8[1]);
        handle_beacon(p);
      } else {
 if(is_my_address(&(p->header.nxh))){
   switch (p->header.typ){
           case DATA:
           PRINTF("[PHD]: Data\n");
           handle_data(p);
           break;
           case RESPONSE:
           PRINTF("[PHD]: Response\n");
           handle_response(p);
           break;
           case OPEN_PATH:
           PRINTF("[PHD]: Open Path\n");
           handle_open_path(p);
```

```
            break;
            case CONFIG:
            PRINTF("[PHD]: Config\n");
            handle_config(p);
            break;
            case REQUEST:
     PRINTF("[PHD]: Request\n");
     handle_request(p);
            break;
            default:
            PRINTF("[PHD]: Report\n");
            handle_report(p);
            break;
          }
 }
 else
  match_packet(p);
      }
    } else {
      packet_deallocate(p);
    }
  }
  void
  handle_beacon(packet_t* p)
  {
   add_neighbor(&(p->header.src),p->info.rssi, 1);
#if !SINK
    PRINTF("Before: Hops = %d NXH: %d.%d\n", conf.hops_from_sink, conf.
        nxh_vs_sink.u8[0], conf.nxh_vs_sink.u8[1]);
    uint8_t new_hops = get_payload_at(p, BEACON_HOPS_INDEX);
    PRINTF("New Hops: %d New RSSI: %d Existing RSSI: %d\n", new_hops, p->info.
        rssi, conf.rssi_from_sink);
    if (new_hops < conf.hops_from_sink-1 || (new_hops == conf.hops_from_sink
        -1 &&
 p->info.rssi < conf.rssi_from_sink))
    {
      conf.nxh_vs_sink = p->header.src;
      conf.hops_from_sink = new_hops+1;
      conf.rssi_from_sink = p->info.rssi;
      conf.sink_address = p->header.nxh;
      PRINTF("After: Hops = %d NXH: %d.%d\n", conf.hops_from_sink, conf.
          nxh_vs_sink.u8[0], conf.nxh_vs_sink.u8[1]);
    }
#endif
    packet_deallocate(p);
```

```
  }
 void
  handle_data(packet_t* p)
  {
    if (is_my_address(&(p->header.dst)))
    {
      PRINTF("Data Packet %d from %d.%d Arrived\n", get_payload_at(p,5), p->
          header.src.u8[0], p->header.src.u8[1]);
      PRINTF("[PHD]: Consuming Packet\n");
      packet_deallocate(p);
    } else {
      match_packet(p);
    }
  }
  void
  handle_report(packet_t* p)
  {
#if SINK
    PRINTF("I got a Report, Sending To Controller From Sink\n");
    send_report_to_controller(p);
#else
    p->header.nxh = conf.nxh_vs_sink;
    rf_unicast_send(p);
#endif
  }
  void
  handle_request(packet_t* p)
  {
#if SINK
    PRINTF("I got a Request, Sending Request to Controller\n");
 send_request_to_controller(p);
#else
    p->header.nxh = conf.nxh_vs_sink;
    rf_unicast_send(p);
#endif
  }
 void
  handle_response(packet_t* p)
  {
    if (is_my_address(&(p->header.dst)))
    {
      entry_t* e = get_entry_from_array(p->payload, p->header.len - PLD_INDEX
          );
      if (e != NULL)
      {
```

```
      add_entry(e);
    }
    packet_deallocate(p);
  } else {
    match_packet(p);
  }
}
void
handle_open_path(packet_t* p)
{
  int i;
  uint8_t n_windows = get_payload_at(p,OPEN_PATH_WINDOWS_INDEX);
  uint8_t start = n_windows*WINDOW_SIZE + 1;
  uint8_t path_len = (p->header.len - (start + PLD_INDEX))/ADDRESS_LENGTH;
  uint8_t my_index = 0;
  uint8_t my_position = 0;
  uint8_t end = p->header.len - PLD_INDEX;
  for (i = start; i < end; i += ADDRESS_LENGTH)
  {
    address_t tmp = get_address_from_array(&(p->payload[i]));
    if (is_my_address(&tmp))
    {
      my_index = i;
      break;
    }
    my_position++;
  }
  if (my_position > 0)
  {
    uint8_t prev = my_index - ADDRESS_LENGTH;
    uint8_t first = start;
    entry_t* e = create_entry();
    window_t* w = create_window();
    w->operation = EQUAL;
    w->size = SIZE_2;
    w->lhs = DST_INDEX;
    w->lhs_location = PACKET;
    w->rhs = MERGE_BYTES(p->payload[first], p->payload[first+1]);
    w->rhs_location = CONST;
    add_window(e,w);
    for (i = 0; i<n_windows; ++i)
    {
      add_window(e, get_window_from_array(&(p->payload[i*WINDOW_SIZE + 1])))
          ;
    }
```

```
    action_t* a = create_action(FORWARD_U, &(p->payload[prev]),
        ADDRESS_LENGTH);
    add_action(e,a);
    PRINTF("[PHD]: ");
    print_entry(e);
    PRINTF("\n");
    add_entry(e);
  }
  if (my_position < path_len-1)
  {
    uint8_t next = my_index + ADDRESS_LENGTH;
    uint8_t last = end - ADDRESS_LENGTH;
    entry_t* e = create_entry();
    window_t* w = create_window();
    w->operation = EQUAL;
    w->size = SIZE_2;
    w->lhs = DST_INDEX;
    w->lhs_location = PACKET;
    w->rhs = MERGE_BYTES(p->payload[last], p->payload[last+1]);
    w->rhs_location = CONST;
    add_window(e,w);
    for (i = 0; i<n_windows; ++i)
    {
      add_window(e, get_window_from_array(&(p->payload[i*WINDOW_SIZE + 1])))
          ;
    }
    action_t* a = create_action(FORWARD_U, &(p->payload[next]),
        ADDRESS_LENGTH);
    add_action(e,a);
    PRINTF("[PHD]: ");
    print_entry(e);
    PRINTF("\n");
    add_entry(e);
    address_t next_address = get_address_from_array(&(p->payload[next]));
    p->header.nxh = next_address;
    p->header.dst = next_address;
    rf_unicast_send(p);
  }
  if (my_position == path_len-1){
    packet_deallocate(p);
  }
 }
void
 handle_config(packet_t* p)
 {
```

```
if (is_my_address(&(p->header.dst))){
 if(p->payload[2] == 114 && p->payload[3] == 102){ //rf - remove
     flowtable
  remove_flowtable();
 }
 else if(p->payload[2] == 115 && p->payload[3] == 102){ //sf - show
     flowtable
       PRINTF("Flow Table\n");
  print_flowtable();
    }
 else if(p->payload[2] == 116 && p->payload[3] == 102 && p->payload[4] ==
      114){ //tfr - turn off radio
  NETSTACK_MAC.off(0);
  PRINTF("Radio Turned Off\n");
 }
 else if(p->payload[2] == 116 && p->payload[3] == 111 && p->payload[4] ==
      114){ //tor - turn on radio
  NETSTACK_MAC.on();
  PRINTF("Radio Turned On\n");
 }
    else{
  PRINTF("Flow Table - Before\n");
  print_flowtable();
  entry_t* e = create_entry();
  action_t* a;
       uint8_t addr[ADDRESS_LENGTH];
       uint8_t addr2[ADDRESS_LENGTH];
  addr[0] = ((p->payload[3] % 10) + 3) % 10;
  addr[1] = 0;
  addr2[0] = ((p->payload[4] % 10) + 3) % 10;
  addr2[1] = 0;
     if(p->payload[2] == 117){ //'u'
      if(p->payload[3] != p->payload[4]){
               a = create_action(FORWARD_U, &(addr2[0]), ADDRESS_LENGTH);
      }
      else{
       a = create_action(FORWARD_U, &(addr[0]), ADDRESS_LENGTH);
      }
      add_action(e,a);
      window_t* w = create_window();
      w->operation = EQUAL;
      w->size = SIZE_2;
      w->lhs = DST_INDEX;
      w->lhs_location = PACKET;
      w->rhs = MERGE_BYTES(addr[0], addr[1]);
```

```
          w->rhs_location = CONST;
          add_window(e,w);
          PRINTF("This Entry to be added to flowtable\n");
          print_entry(e);
          add_entry(e);
          PRINTF("Flow Table - After\n");
          print_flowtable();
         }
         else if(p->payload[2] == 98){ //'b'
          uint8_t newaddr[ADDRESS_LENGTH];
            newaddr[0] = newaddr[1] = 255;
          a = create_action(FORWARD_B, &(newaddr[0]), ADDRESS_LENGTH);
         }
        }
     packet_deallocate(p);
 }
 else
  match_packet(p);
  }
 void
  test_handle_open_path(void)
  {
    uint8_t array[19] = {
      1, 19, 0, 1, 0, 2, 5, 100, 0, 1, 0, 0, 1, 0, 2, 0, 3, 0, 4,
    };
    packet_t* p = get_packet_from_array(array);
    handle_open_path(p);
  }
```

A.2.4 packet-buffer.c

```
#include <string.h>
#include <stdio.h>
#include "lib/memb.h"
#include "lib/list.h"
#include "packet-buffer.h"
#include "address.h"
#define MAX_TTL 100
#define DEBUG 1
#if DEBUG && (!SINK || DEBUG_SINK)
#define PRINTF(...) printf(__VA_ARGS__)
#else
#define PRINTF(...)
#endif
MEMB(packets_memb, packet_t, 4);
  static packet_t * packet_allocate(void);
```

```
 void print_report_data(uint8_t a, uint8_t b, uint8_t c, uint8_t d){
   printf("D#Million: Data Command d%u%u%u%u\n", a,b,c,d);
 }
 void print_report_config(uint8_t a, uint8_t b, uint8_t c, uint8_t d){
   printf("C#Million: Config Command %u%uu%u%u\n", a,b,c,d);
 }
 void
 print_packet_uart(packet_t* p)
 {
   uint16_t i = 0;
   packet_deallocate(p);
 }
 void
 send_report_to_controller(packet_t* p)
 {
PRINTF("Inside Send_report_to_controller function\n");
printf("Report:\n");
   uint8_t len = p->payload[2];
uint8_t reportlen = len*3 + 2;
uint8_t report[reportlen];
int k=0;
report[0] = p->header.src.u8[0];
report[1] = p->header.src.u8[1];
      for(k=2;k<reportlen;k++){
 report[k] = p->payload[k+1];
}
int j = 0;
printf("%d",report[j]);
for(j=1;j<reportlen;j++){
 printf(",");
 printf("%d",report[j]);
}
printf("\n");
packet_deallocate(p);
 }
 void
 send_request_to_controller(packet_t* p)
 {
   PRINTF("Inside Send_request_to_controller function\n");
   printf("Request:\n");
   uint8_t request[4];
   request[0] = p->header.src.u8[0];
   request[1] = p->header.src.u8[1];
   request[2] = p->payload[5];
request[3] = p->payload[6];
```

```
int j=0;
  printf("%d",request[j]);
  for(j=1;j<4;j++){
      printf(",");
      printf("%d",request[j]);
  }
  printf("\n");
  packet_deallocate(p);
}
void
print_packet(packet_t* p)
{
  uint16_t i = 0;
  PRINTF("Network ID: %d Packet Length: %d ", p->header.net, p->header.len)
      ;
  PRINTF("Packet Dst:");
  print_address(&(p->header.dst));
  PRINTF("Packet Src:");
  print_address(&(p->header.src));
  PRINTF("Pkt Type: %d TTL: %d ", p->header.typ, p->header.ttl);
  PRINTF("Next Hop: ");
  print_address(&(p->header.nxh));
  PRINTF("Payload: ");
  for (i=0; i < (p->header.len - PLD_INDEX); ++i){
    PRINTF("%d ",get_payload_at(p,i));
  }
}
static packet_t *
packet_allocate(void)
{
  packet_t *p = NULL;
  p = memb_alloc(&packets_memb);
  if(p == NULL) {
    PRINTF("[PBF]: Failed to allocate a packet\n");
  }
  return p;
}
void
packet_deallocate(packet_t* p)
{
  int res = memb_free(&packets_memb, p);
  if (res !=0){
    PRINTF("[FLT]: Failed to deallocate a packet. Reference count: %d\n",
        res);
  }
```

```
}
packet_t*
create_packet_payload(uint8_t net, address_t* dst, address_t* src,
  packet_type_t typ, address_t* nxh, uint8_t* payload, uint8_t len)
{
  packet_t* p = create_packet(net, dst, src, typ, nxh);
  if (p != NULL){
    uint8_t i;

    for (i = 0; i < len; ++i){
      set_payload_at(p, i, payload[i]);
    }
  }
  return p;
}
packet_t*
get_packet_from_array(uint8_t* array)
{
  address_t dst = get_address_from_array(&array[DST_INDEX]);
  address_t src = get_address_from_array(&array[SRC_INDEX]);
  address_t nxh = get_address_from_array(&array[NXH_INDEX]);
  packet_t* p = create_packet_payload(array[NET_INDEX], &dst, &src,
    array[TYP_INDEX], &nxh, &array[PLD_INDEX], array[LEN_INDEX] - PLD_INDEX
        );
  return p;
}
uint8_t
get_payload_at(packet_t* p, uint8_t index)
{
  if (index < MAX_PACKET_LENGTH){
    return p->payload[index];
  } else {
    return 0;
  }
}
void
set_payload_at(packet_t* p, uint8_t index, uint8_t value)
{
  if (index < MAX_PACKET_LENGTH){
    p->payload[index] = value;
    if (index + PLD_INDEX + 1 > p->header.len){
      p->header.len = index + PLD_INDEX + 1;
    }
  }
}
```

```
void
restore_ttl(packet_t* p)
{
  p->header.ttl = MAX_TTL;
}
packet_t*
create_packet_empty(void)
{
  packet_t* p = packet_allocate();
  if (p != NULL){
    memset(&(p->header), 0, sizeof(p->header));
    memset(&(p->info), 0, sizeof(p->info));
    restore_ttl(p);
  }
  return p;
}
packet_t*
create_packet(uint8_t net, address_t* dst, address_t* src, packet_type_t
    typ,
  address_t* nxh)
{
  packet_t* p = packet_allocate();
  if (p != NULL){
    memset(&(p->header), 0, sizeof(p->header));
    memset(&(p->info), 0, sizeof(p->info));
    p->header.net=net;
    p->header.dst=*dst;
    p->header.src=*src;
    p->header.typ=typ;
    p->header.nxh=*nxh;
    restore_ttl(p);
  }
  return p;
}
void
packet_buffer_init(void)
{
  memb_init(&packets_memb);
}
void
test_packet_buffer(void)
{
  uint8_t array[73] = {1, 73, 0, 0, 0, 2, 4, 100, 0, 0, 20, 18, 0, 6, 0,
      10,
```

```
    18, 0, 50, 0, 1, 90, 0, 10, 0, 1, 122, 0, 12, 0, 5, 1, 4, 8, 6, 2, 0,
        10,
    0, 40, 0, 0, 8, 5, 1, 0, 0, 0, 0, 0, 0, 1, 3, 3, 2, 255, 255, 3, 1, 0,
        3,
    1, 7, 8, 6, 132, 0, 11, 0, 12, 0, 13, 254};

    packet_t* second = get_packet_from_array(array);
    print_packet(second);
}
```

A.2.5 node-conf.c

```
#include <string.h>
#include "address.h"
#include "node-conf.h"
#include "net/rime/rime.h"
#define _MY_ADDRESS 1
#define _NET 1
#define _BEACON_PERIOD 5
#define _REPORT_PERIOD 20
#define _RULE_TTL 100
#define _RSSI_MIN 0
#define _PACKET_TTL 100;
#define DEBUG 1
#if DEBUG && (!SINK || DEBUG_SINK)
#include <stdio.h>
#define PRINTF(...) printf(__VA_ARGS__)
#else
#define PRINTF(...)
#endif
  node_conf_t conf;
  void
  node_conf_init(void)
  {
#if COOJA
    conf.my_address.u8[1] = linkaddr_node_addr.u8[0];
    conf.my_address.u8[0] = linkaddr_node_addr.u8[1];
#endif
    conf.requests_count = 0;
    conf.my_net = _NET;
    conf.beacon_period = _BEACON_PERIOD;
    conf.report_period = _REPORT_PERIOD;
    conf.rule_ttl = _RULE_TTL;
    conf.rssi_min = _RSSI_MIN;
    conf.packet_ttl = _PACKET_TTL;
#if SINK
```

```
    conf.my_address.u8[0] = 1;
    conf.my_address.u8[1] = 0;
    conf.is_active = 1;
    conf.nxh_vs_sink = conf.my_address;
    conf.sink_address = conf.my_address;;
    conf.hops_from_sink = 0;
    conf.rssi_from_sink = 0;
#endif
#if NODE1
    conf.my_address.u8[0] = 2;
    conf.my_address.u8[1] = 0;
    conf.sink_address.u8[0] = 1;
    conf.sink_address.u8[1] = 0;
    conf.nxh_vs_sink = conf.sink_address;
    conf.is_active = 0;
    conf.hops_from_sink = _PACKET_TTL;
    conf.rssi_from_sink = 100;
#endif
#if NODE2
    conf.my_address.u8[0] = 3;
    conf.my_address.u8[1] = 0;
    conf.sink_address.u8[0] = 1;
    conf.sink_address.u8[1] = 0;
    conf.nxh_vs_sink = conf.sink_address;
    conf.is_active = 0;
    conf.hops_from_sink = _PACKET_TTL;
    conf.rssi_from_sink = 100;
#endif
#if NODE3
    conf.my_address.u8[0] = 4;
    conf.my_address.u8[1] = 0;
    conf.sink_address.u8[0] = 1;
    conf.sink_address.u8[1] = 0;
    conf.nxh_vs_sink = conf.sink_address;
    conf.is_active = 0;
    conf.hops_from_sink = _PACKET_TTL;
    conf.rssi_from_sink = 100;
#endif
#if NODE4
    conf.my_address.u8[0] = 5;
    conf.my_address.u8[1] = 0;
    conf.sink_address.u8[0] = 1;
    conf.sink_address.u8[1] = 0;
    conf.nxh_vs_sink = conf.sink_address;
    conf.is_active = 0;
```

```
    conf.hops_from_sink = _PACKET_TTL;
    conf.rssi_from_sink = 100;
#endif
#if NODE5
    conf.my_address.u8[0] = 6;
    conf.my_address.u8[1] = 0;
    conf.sink_address.u8[0] = 1;
    conf.sink_address.u8[1] = 0;
    conf.nxh_vs_sink = conf.sink_address;
    conf.is_active = 0;
    conf.hops_from_sink = _PACKET_TTL;
    conf.rssi_from_sink = 100;
#endif
#if NODE6
    conf.my_address.u8[0] = 7;
    conf.my_address.u8[1] = 0;
    conf.sink_address.u8[0] = 1;
    conf.sink_address.u8[1] = 0;
    conf.nxh_vs_sink = conf.sink_address;
    conf.is_active = 0;
    conf.hops_from_sink = _PACKET_TTL;
    conf.rssi_from_sink = 100;
#endif
  }
  void
  print_node_conf(void){
    PRINTF("[CFG]: NODE: ");
    print_address(&(conf.my_address));
    PRINTF("\n");
    PRINTF("[CFG]: - Network ID: %d\n[CFG]: - Beacon Period: %d\n[CFG]: - "
      "Report Period: %d\n[CFG]: - Rules TTL: %d\n[CFG]: - Min RSSI: "
      "%d\n[CFG]: - Packet TTL: %d\n[CFG]: - Next Hop -> Sink: ",
      conf.my_net, conf.beacon_period, conf.report_period,
      conf.rule_ttl, conf.rssi_min, conf.packet_ttl);
    print_address(&(conf.nxh_vs_sink));
    PRINTF(" (hops: %d, rssi: %d)\n", conf.hops_from_sink, conf.
        rssi_from_sink);
    PRINTF("[CFG]: - Sink: ");
    print_address(&(conf.sink_address));
    PRINTF("\n");
  }
```

Appendix B
Experimental Measured Data

B.1 Convergence Time Measurements

The measured convergence time for the first three scenarios is shown below in table B.1. All values are in seconds.

Instant of Run	Single Hop Mesh Topology	Multi Hop Topology 1	Multi Hop Topology 2
1	19.788609	36.894394	41.766896
2	19.757513	41.690633	62.342307
3	19.541923	41.194494	41.510856
4	19.914289	57.236108	42.451018
5	19.903602	44.868333	47.266991
6	19.930099	41.611021	42.412098
7	19.949932	66.952811	41.927383
8	19.956068	37.081049	42.378835
9	18.382732	35.671318	41.734585
10	20.518535	41.665969	38.078382
11	20.059682	37.986977	38.606651
12	20.687239	41.429538	37.342341
13	19.931188	42.01233	35.657711
14	20.301484	36.961322	41.590414
15	20.070659	36.407952	42.554282
16	20.439672	37.176165	47.383563
17	20.472519	37.331236	53.472209
18	20.105869	36.592075	47.731189
19	18.919486	35.531434	42.207012
20	20.374409	36.985215	72.603565

Table B.1: Convergence Time of Different Topologies

B.2 Packet Delay Measurements

B.2.1 Scenario 1: Single Hop Mesh Topology

Packet delay measurement from Node 1 to the other Nodes in the single hop mesh topology scenario (Figure 4.2) is given below in table B.2. All values are in seconds.

Instant of Run	Node 1 to 2	Node 1 to 3	Node 1 to 4	Node 1 to 5	Node 1 to 6
1	0.19	0.181	0.174	0.15	0.356
2	0.184	0.178	0.169	0.145	0.227
3	0.179	0.296	0.166	0.141	0.221
4	0.176	0.293	0.159	0.135	0.218
5	0.171	0.288	0.781	0.128	0.211
6	0.165	0.284	0.151	0.25	0.209
7	0.161	0.283	0.146	0.243	0.204
8	0.153	0.275	0.14	0.239	0.198
9	0.146	0.273	0.135	0.234	0.192
10	0.517	0.144	0.125	0.228	0.188
11	0.137	0.142	0.112	0.223	0.308
12	0.254	0.137	0.199	0.218	0.163
13	0.246	0.132	0.186	0.211	0.16
14	0.245	0.126	0.18	0.207	0.154
15	0.239	0.127	0.175	0.203	0.15
16	0.22	0.121	0.17	0.199	0.144
17	0.215	0.102	0.165	0.176	0.265
18	0.211	0.564	0.535	0.171	0.262
19	0.206	0.183	0.155	0.165	0.254
20	0.201	0.18	0.152	0.16	0.251
21	0.206	0.174	0.146	0.155	0.245
22	0.19	0.17	0.138	0.149	0.24
23	0.188	0.164	0.618	0.144	0.238
24	0.181	0.161	0.114	0.139	0.231
25	0.177	0.151	0.109	0.211	0.227
26	0.172	0.131	0.103	0.253	0.222
27	0.167	0.126	0.098	0.247	0.217
28	0.164	0.996	0.093	0.243	0.213
29	0.156	0.113	0.204	0.363	0.193
30	0.15	0.108	0.193	0.233	0.188

31	0.148	0.103	0.314	0.228	0.184
32	0.377	0.079	0.185	0.216	0.176
33	0.371	0.192	0.18	0.446	0.3
34	0.733	0.19	0.175	0.19	0.168
35	0.343	0.106	0.21	0.183	0.163
36	0.158	0.214	0.208	1.303	0.159
37	0.157	0.212	0.202	0.173	0.153
38	0.158	0.211	0.201	0.168	0.274
39	0.156	0.206	0.195	0.163	0.269
40	0.155	0.202	0.19	0.159	0.139
41	0.232	0.188	0.175	0.152	0.261
42	0.222	0.169	0.298	0.147	0.254
43	0.131	0.287	0.168	0.141	0.267
44	0.126	0.155	0.163	0.386	0.264
45	0.125	0.134	0.157	0.131	0.258
46	0.123	0.132	0.152	0.125	0.13
47	0.122	0.126	0.148	0.139	0.124
48	0.137	0.12	0.253	0.133	0.119
49	0.252	0.115	0.241	0.128	0.18
50	0.246	0.104	0.229	0.123	0.176
51	0.114	0.103	0.226	0.119	0.169
52	0.144	0.222	0.221	0.112	0.164
53	0.141	0.213	0.215	0.11	0.159
54	0.141	0.208	0.21	0.104	0.154
55	0.31	0.577	0.197	0.102	0.15
56	0.29	0.198	0.182	0.21	0.143
57	0.161	0.181	0.16	0.082	0.139
58	0.282	0.8	0.277	0.199	0.135
59	0.277	0.164	0.275	0.173	0.193
60	0.257	0.161	0.269	0.168	0.315
61	0.378	0.151	0.263	0.163	0.184
62	0.249	0.147	0.259	0.158	0.181
63	0.371	0.268	0.252	0.153	0.175
64	0.24	0.509	0.245	0.147	0.17
65	0.237	0.13	0.24	0.143	0.165
66	0.23	0.199	0.987	0.132	0.159

67	0.353	0.19	0.231	0.116	0.154
68	0.341	0.184	0.217	0.111	0.15
69	0.214	0.42	0.203	0.106	0.143
70	0.203	0.167	0.193	0.098	0.229
71	0.198	0.551	0.187	0.095	0.224
72	0.196	0.422	0.181	0.16	0.222
73	0.304	0.167	0.428	0.155	0.215
74	0.173	0.165	0.174	0.154	0.586
75	0.171	0.161	0.168	0.151	0.206
76	0.165	0.531	0.166	0.147	0.201
77	0.407	0.152	0.16	0.14	0.198
78	0.278	0.146	0.156	0.136	0.441
79	0.271	0.14	0.152	0.131	0.187
80	0.268	0.138	0.269	0.125	0.307
81	0.262	0.133	0.263	0.081	0.431
82	0.257	0.115	0.254	0.441	0.165
83	0.254	0.363	0.244	0.174	0.163
84	0.247	0.107	0.238	0.308	0.153
85	0.238	0.101	0.235	0.428	0.15
86	0.236	0.097	0.229	0.422	0.145
87	0.212	0.094	0.975	0.417	0.268
88	0.21	0.158	0.222	0.411	0.263
89	0.194	0.159	0.215	0.406	0.257
90	0.191	0.15	0.201	0.647	0.255
91	0.186	0.276	0.196	0.636	0.248
92	0.179	0.151	0.187	0.652	0.245
93	0.176	0.15	0.282	0.224	0.239
94	0.171	0.147	0.278	0.583	0.234
95	0.167	0.271	0.259	0.198	0.231
96	0.162	0.267	0.253	0.192	0.227
97	0.156	0.266	0.25	0.57	0.223
98	0.206	0.264	0.235	0.805	0.221
99	0.206	0.263	0.23	0.177	0.207
100	0.204	0.262	0.721	0.794	0.202

Table B.2: Single Hop Mesh Topology Packet Delay Measurements

B.2.2 Scenario 2: Multi Hop Topology 1

Packet delay measurement from Node 1 to the other Nodes in the multi hop scenario 2 topology (Figure 4.3) is given below in table B.3. All values are in seconds.

Instant of Run	Node 1 to 2	Node 1 to 3	Node 1 to 4	Node 1 to 5	Node 1 to 6
1	0.169	0.34	0.586	0.439	0.385
2	0.163	0.336	0.576	0.437	0.379
3	0.156	0.335	0.448	0.433	0.376
4	0.394	0.197	0.564	0.423	0.374
5	0.128	0.194	0.482	0.917	0.371
6	0.123	0.433	0.479	0.413	0.365
7	0.241	0.299	0.46	0.409	0.362
8	0.109	0.419	0.378	0.512	0.355
9	0.106	0.78	0.5	0.757	0.352
10	0.47	0.268	0.409	1.003	0.347
11	0.22	0.643	0.277	0.492	0.702
12	0.213	0.381	0.277	0.367	0.326
13	0.209	0.253	0.272	0.358	0.321
14	0.576	0.253	0.644	0.356	0.317
15	0.179	0.495	0.266	0.976	0.312
16	0.119	0.237	0.639	0.347	0.311
17	0.74	0.235	0.379	0.345	0.307
18	0.11	0.48	0.25	0.841	0.677
19	0.206	0.217	0.619	0.341	0.671
20	0.199	0.341	0.241	0.585	0.791
21	0.188	0.209	0.236	0.831	0.535
22	0.186	0.204	0.338	0.451	0.535
23	0.156	0.185	0.208	0.681	0.531
24	0.136	0.433	0.329	0.677	0.649
25	0.258	0.177	0.573	0.42	0.894
26	0.244	0.673	0.318	0.792	0.501
27	0.239	0.282	0.313	0.914	0.624
28	0.606	0.406	0.432	0.91	0.747
29	0.32	0.274	0.556	0.531	0.492
30	0.19	0.511	0.299	0.525	0.612
31	0.177	0.631	0.294	0.772	0.857
32	0.154	0.25	0.29	0.514	0.603

33	0.151	0.247	0.289	0.637	0.598
34	0.515	0.463	0.287	0.746	0.721
35	0.263	0.449	0.254	0.495	0.965
36	0.133	0.571	0.374	0.992	0.586
37	0.248	0.431	0.246	0.859	0.583
38	0.148	0.666	0.241	0.481	0.827
39	0.268	0.41	0.36	0.475	0.577
40	0.14	0.404	0.606	0.472	0.41
41	0.131	0.54	0.35	0.589	0.405
42	0.75	0.652	0.221	0.459	0.4
43	0.12	0.647	0.341	0.455	0.398
44	0.74	0.656	0.338	0.576	0.397
45	0.102	0.492	0.332	0.444	0.394
46	0.093	0.634	0.575	0.44	0.392
47	0.211	0.628	0.568	0.434	0.391
48	0.192	0.619	0.314	0.429	0.388
49	0.187	0.614	0.567	0.424	0.492
50	0.185	0.462	0.434	0.42	0.491
51	0.18	0.461	0.544	0.415	0.488
52	0.179	0.604	0.539	0.524	0.487
53	0.174	0.594	0.494	0.396	0.484
54	0.167	0.317	0.357	0.391	0.481
55	0.165	0.298	0.349	0.389	0.478
56	0.16	0.562	0.595	0.384	0.474
57	0.154	0.557	0.341	0.487	0.469
58	0.149	0.547	0.463	0.484	0.463
59	0.138	0.542	0.333	0.479	0.964
60	0.135	0.391	0.328	0.477	0.455
61	0.122	0.386	0.694	0.475	0.574
62	0.225	0.755	0.439	0.47	0.444
63	0.215	0.526	0.674	0.465	0.445
64	0.21	0.519	0.542	0.46	0.436
65	0.205	0.367	0.762	0.454	0.435
66	0.197	0.376	0.606	0.449	0.43
67	0.192	0.501	0.221	0.444	0.425
68	0.187	0.718	0.57	0.441	0.419

69	0.173	0.297	0.444	0.436	0.419
70	0.167	0.293	0.541	0.392	0.402
71	0.152	0.29	0.789	0.387	0.4
72	0.224	0.286	0.49	0.387	0.401
73	0.223	0.404	0.485	0.383	0.398
74	0.213	0.399	0.454	0.38	0.393
75	0.204	0.395	0.568	0.498	0.389
76	0.321	0.389	0.687	0.37	0.507
77	0.318	0.384	0.519	0.365	0.506
78	0.555	0.382	0.594	0.358	0.497
79	0.309	0.376	0.335	0.352	0.495
80	0.174	0.369	0.333	0.348	0.495
81	0.171	0.363	0.582	0.339	0.488
82	0.542	0.344	0.456	0.314	0.449
83	0.288	0.338	0.329	0.55	0.448
84	0.144	0.334	0.452	0.419	0.568
85	0.765	0.328	0.327	0.414	0.439
86	0.254	0.323	0.323	0.409	0.434
87	0.374	0.321	0.444	0.404	0.434
88	0.353	0.433	0.307	0.4	0.432
89	0.718	0.305	0.306	0.389	0.43
90	0.456	0.301	0.301	0.384	0.426
91	0.303	0.295	0.301	0.381	0.424
92	0.544	0.417	0.55	0.376	0.418
93	0.161	0.411	0.423	0.358	0.416
94	0.527	0.406	0.422	0.356	0.411
95	0.648	0.401	0.317	1.225	0.407
96	0.272	0.396	0.316	0.344	0.4
97	0.518	0.392	0.314	0.343	0.512
98	0.139	0.374	0.313	0.337	0.507
99	0.508	0.37	0.313	0.33	0.501
100	0.26	0.364	0.312	0.325	0.499

Table B.3: Multi Hop Topology 1 Packet Delay Measurements

B.2.3 Scenario 3: Multi Hop Topology 2

Packet delay measurement from Node 1 to the other Nodes in the multi hop scenario 3 topology (Figure 4.4) is given below in table B.4. All values are in seconds.

Instant of Run	Node 1 to 2	Node 1 to 3	Node 1 to 4	Node 1 to 5	Node 1 to 6
1	0.288	0.348	1.531	0.641	0.669
2	0.153	0.34	0.401	0.76	0.663
3	0.274	0.338	0.646	0.382	0.783
4	0.143	0.335	0.389	2.874	0.65
5	0.134	0.327	0.512	1.744	0.962
6	0.095	0.32	0.381	1.739	1.457
7	0.217	0.315	0.732	1.613	0.52
8	0.213	0.31	0.353	0.985	0.795
9	0.203	0.299	0.471	0.477	0.54
10	0.197	0.296	0.467	1.097	0.523
11	0.192	0.29	0.465	2.217	0.768
12	0.189	0.285	0.335	0.581	0.633
13	0.433	0.281	0.455	0.701	0.989
14	0.179	0.272	0.45	0.929	0.611
15	0.549	0.267	0.444	0.549	1.356
16	0.173	0.262	0.44	0.669	0.944
17	0.613	0.259	0.434	0.539	1.18
18	0.207	0.242	0.554	0.907	0.787
19	0.139	0.355	0.802	0.778	1.158
20	0.138	0.352	0.421	1.023	1.402
21	0.387	0.347	1.789	0.643	0.771
22	0.38	0.338	1.287	0.76	1.014
23	0.12	0.333	0.53	0.757	1.508
24	0.234	0.327	0.513	1.752	0.753
25	0.345	0.324	0.501	0.622	1.114
26	0.335	0.307	0.372	1.366	0.854
27	0.414	0.298	0.493	0.985	1.474
28	0.384	0.291	0.364	1.482	1.22
29	0.493	0.281	0.483	0.726	0.708
30	0.14	0.279	0.606	0.719	0.581
31	0.253	0.274	0.601	0.698	0.815
32	0.245	0.266	0.345	0.819	1.447

33	0.25	0.256	0.465	0.812	0.944
34	0.201	0.378	0.586	1.559	0.938
35	0.196	0.373	0.582	0.679	1.434
36	0.189	0.367	0.451	0.671	1.177
37	0.203	0.362	0.447	0.919	0.548
38	0.573	0.357	0.441	0.914	1.045
39	0.502	0.338	0.686	0.531	0.897
40	0.358	0.335	0.544	0.652	1.145
41	0.102	0.328	0.537	0.896	0.639
42	0.598	0.324	0.533	0.641	0.509
43	0.219	0.941	0.528	0.76	1.004
44	0.215	0.811	0.397	0.754	0.621
45	0.433	1.179	0.393	0.75	0.864
46	0.179	0.926	0.389	1.368	0.985
47	0.22	0.92	0.507	0.738	1.481
48	0.198	0.536	0.629	0.467	1.242
49	0.434	0.533	0.376	0.588	0.471
50	0.301	0.513	0.372	0.585	0.832
51	0.299	0.325	0.491	0.578	0.829
52	0.293	0.322	0.361	0.699	0.681

Table B.4: Multi Hop Scenario 2 Topology Packet Delay Measurements

B.3 Packet Loss Measurements

B.3.1 Scenario 1: Single Hop Mesh Topology

Instant of Run	Node 1 to 2	Node 1 to 3	Node 1 to 4	Node 1 to 5	Node 1 to 6
1	2	2	2	7	3
2	2	2	2	2	6
3	2	2	2	2	2
4	2	2	2	2	6
5	4	2	2	2	2
6	2	2	4	3	2
7	6	2	2	2	2
8	2	2	2	2	2

Table B.5: Single Hop Mesh Topology Packet Loss Measurements

B.3.2 Scenario 2: Multi Hop Topology 1

Instant of Run	Node 1 to 2	Node 1 to 3	Node 1 to 4	Node 1 to 5	Node 1 to 6
1	3	4	11	10	21
2	2	4	4	10	11
3	2	5	5	10	11
4	2	4	4	10	10
5	2	5	4	8	17
6	2	6	5	10	13
7	2	7	4	10	10
8	3	4	5	10	25

Table B.6: Multi Hop Topology 1 Packet Loss Measurements

B.3.3 Scenario 3: Multi Hop Topology 2

Instant of Run	Node 1 to 2	Node 1 to 3	Node 1 to 4	Node 1 to 5	Node 1 to 6
1	2	4	10	35	70
2	3	4	17	32	69
3	4	5	13	58	56
4	2	5	10	37	27
5	2	4	16	50	79
6	2	4	11	62	42
7	2	4	11	63	64
8	3	4	11	19	32

Table B.7: Multi Hop Topology 2 Packet Loss Measurements

B.4 Control Message Overhead Measurements

B.4.1 Periodic Local Topology Report Messages

All the measurements are per minute.

Instant of Run	Single Hop Topology	Multi Hop Topology 1	Multi Hop Topology 2
1	19	21	23
2	19	21	21
3	18	21	20
4	18	20	20

5	21	22	20
6	21	20	20
7	21	20	21
8	21	21	19
9	21	18	19
10	21	17	20
11	21	19	19
12	21	21	19
13	21	18	18
14	20	21	19
15	19	21	20
16	19	18	21
17	18	19	21
18	20	23	21
19	19	17	21
20	20	19	21

Table B.8: Local Topology Report Message Overhead Every Minute

B.4.2 Flow Request and Response Message Overhead

Scenario 1: Single Hop Mesh Topology

Instant of Run	Node 1 to 2	Node 1 to 3	Node 1 to 4	Node 1 to 5	Node 1 to 6
1	8	8	8	8	8
2	8	8	8	8	18
3	8	8	8	8	10
4	8	8	8	8	16
5	14	8	8	8	8
6	8	8	14	10	8
7	10	8	8	6	12

Table B.9: Control Message Overhead in Single Hop Mesh Topology

Scenario 2: Multi Hop Topology 1

Instant of Run	Node 1 to 2	Node 1 to 3	Node 1 to 4	Node 1 to 5	Node 1 to 6
1	8	18	16	34	36

2	8	18	16	34	36
3	8	12	16	36	52
4	8	16	16	36	36
5	8	16	18	30	36
6	8	16	16	38	34
7	6	16	18	52	34

Table B.10: Control Message Overhead in Multi Hop Topology 1

Scenario 3: Multi Hop Topology 2

Instant of Run	Node 1 to 2	Node 1 to 3	Node 1 to 4	Node 1 to 5	Node 1 to 6
1	10	16	30	122	228
2	8	14	34	80	270
3	6	18	36	182	208
4	8	18	40	156	58
5	8	18	34	344	314
6	8	16	38	236	66
7	8	16	36	206	128

Table B.11: Control Message Overhead in Multi Hop Topology 2

B.5 Rule Installation Time Measurements

Instant of Measurement	Requested At	Flow Installed At	Time Difference (sec)
1	33.586458	33.806	0.219542
2	3.586509	3.811	0.224491
3	50.039554	50.255	0.215446
4	28.873922	29.059	0.185078
5	58.875206	59.058	0.182794
6	9.925826	10.146	0.220174
7	40.051549	40.272	0.220451
8	39.256649	39.444	0.187351
9	49.383525	49.569	0.185475
10	23.236809	23.406	0.169191
11	53.237234	53.406	0.168766
12	20.057912	20.22	0.162088

13	27.857713	28.025	0.167287
14	45.483599	45.65	0.166401
15	25.413195	25.605	0.191805
16	4.538683	4.732	0.193317
17	48.760891	48.986	0.225109
18	7.756465	8.36	0.603535
19	48.59922	48.882	0.28278
20	6.351056	6.634	0.282944
21	34.645874	34.868	0.222126
22	52.770715	52.994	0.223285
23	34.904684	35.074	0.169316
24	52.651553	53.199	0.547447
25	5.736072	5.931	0.194928
26	13.751071	13.938	0.186929
27	30.258577	30.44	0.181423
28	58.301662	58.44	0.138338
29	56.445463	56.623	0.177537
30	35.73768	35.915	0.17732
31	43.752742	43.92	0.167258
32	41.03416	41.607	0.57284
33	37.113188	37.28	0.166812
34	36.974499	37.173	0.198501
35	51.975595	52.174	0.198405
36	30.416898	30.667	0.250102
37	45.127597	45.344	0.216403
38	53.142842	53.233	0.090158

Table B.12: Rule (Flow Entry) Installation Time Measurements