MASTER IN COMPUTER SCIENCE

SDNWisebed

A Software-Defined Wireless Sensor Network Testbed

Master Thesis

Jakob Schärer

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

July 2018

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Anhang 1 / Umschlagseite

SDNWisebed

A Software-Defined Wireless Sensor Network Testbed

Masterarbeit der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Jakob Schärer

2018

Leiter der Arbeit

Prof. Dr. T. Braun

"The nice thing about standards is that you have so many to choose from."

Andrew S. Tanenbaum

Abstract

Software Defined Networking (SDN) is a promising approach to simplify the management of Wireless Sensor Networks (WSNs). Many SDN frameworks for WSNs have been proposed. However, real-world testbeds to accelerate the development of SDN-based WSN applications are still rare. In this work we propose SDNWisebed a testbed consisting of sensor nodes distributed over two buildings in office-like environments. This testbed was designed to evaluate various types of SDN-based WSN applications, such as WSN routing protocols and network applications, before deploying them in real-world infrastructures. This paper focuses on the integration of the SDN-Wise framework with TARWIS, a testbed management architecture for real-world WSN testbeds. We conducted both functional and performance evaluation. Evaluation results show that with the integration of SDN architecture, performance of WSN applications can be boosted significantly.

Contents

List of Figures

List of Tables

List of Abbreviations

DODAG	**D**estination **O**riented **D**irected **A**cyclic **G**raph
DTARP	**D**ynamic **T**raffic **A**ware **R**outing **P**rotocol
IoT	**I**nternet **o**f **T**hings
ONOS	**O**pen **N**etwork **O**perating **S**ystem
RPL	**R**outing **P**rotocol for **L**ow power and Lossy Networks
SD	**S**mart **D**evice
SDN	**S**oftware **D**efined **N**etworking
SDWSN	**S**oftware-**D**efined **W**ireless **S**ensor **N**etwork
TARWIS	Testbed management architecture for wireless sensor network testbeds
WSN	**W**ireless **S**ensor **N**etwork

Chapter 1

Introduction

It is the vision of the Internet of Things (IoT) that in the near future many smart devices will be interconnected with each other and exchange information to support humans in their daily lives. While it is very fascinating to see how far the integration of this vision progressed during the past two decades, there are still some problems to solve. These days, many smart devices of different manufacturers exist, but the interconnection and management of these devices is still a tedious task. No standard to interconnect different device types with each other has been established and, therefore, the vision of IoT is often restricted to the interconnection of devices of the same manufacturer. Consequently, we are currently still living in the area of Intranets of Things and not yet in the area of Internet of Things. To accelerate the integration of IoT, interfaces that promote the easy management and machine to machine communication of smart devices are needed. Software Defined Networking (SDN) is a promising approach to simplify the management of IoT networks. When the size of traditional networks evolved and the number of network devices of different manufactures exceeded the size of manageability, SDN was very successful in simplifying the configuration of networks with devices of various manufacturers and minimizing the need of human interaction.

Wireless Sensor Networks (WSNs) are a major component of the IoT. In WSNs many sensor nodes are used to collect physical parameters in an area of interest. WSNs have been a subject of broad interest in the past two decades and therefore, they are well studied. But the development of IoT during the past few years, has added traction to WSN research. Instead of only sensing and collecting physical properties in an area of interest the sensor nodes become smarter and can now be seen as smart devices. The state and output of these smart sensors might depend on other smart devices and therefore, communication between those devices must be possible. For example, the thermostat in a smart building should be able to fetch the values of all temperature sensors in the room for that it is controlling the temperature. The routing protocols that have been developed in the first epoch of WSN research are mainly optimized for node to sink communication but inefficient for node to node communication. With Software Defined Networking more sophisticated algorithms that enable efficient node to node communication might be possible, but extensive research on this topic is still missing. Some SDN frameworks for WSNs have already been proposed and their implementations are available, but the network applications are still rare.

SDN-Wise is one of these available and already implemented SDN frameworks. 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 (*SDN-Wise*).

A major reason for the slow integration of new SDN-based applications is the lack of good test environments that enable rapid prototyping of new applications. To design and evaluate new SDN-based applications and routing protocols, reliable, fast and precise testing environments are needed. Simulators like ns-3, OMNeT++ and COOJA can be used for the simulation of WSNs (Riley and Henderson, 2010; Varga, 2010; Osterlind et al., 2006). With these simulations research projects can be tested, evaluated and initially validated. But since these simulators are based on mathematical models, they always need to abstract the underlying physical system. The accuracy of the results always depends on the accuracy of the mathematical model of the simulator. To achieve realistic a precise test results, the network applications also must be tested in a real testbed.

A real world testbed that supports SDN-based applications and enables rapid prototyping, would accelerate the integration of new applications and could help SDN to establish in WSNs. When SDN is more used in WSNs, the interconnection between WSNs simplifies and the adaptation of IoT could make a leap forwards.

1.1 Motivation

The research of WSNs and its applications has been a topic of broad interest in the past two decades. But despite of all these efforts, only a few large scale WSN applications have been deployed into production. The major reason for the slow development of new WSN applications is the high complexity in the management and the deployment of new WSN applications. The characteristics of the WSN networks are very application specific and, therefore, the configuration of a huge number of sensor nodes is tedious, error prone, and complex. SDN is envisioned to solve the problem of the management complexity by centralizing the network management on a controller. This SDN approach is very promising, but only a few WSN-Testbeds that support the evaluation of SDN-based network applications are available. These SDN enabled WSN-Testbeds are urgently needed that the research and development community can focus on the design of new WSN applications.

The few available SDN enabled WSN-Testbeds are mainly designed to test specific applications and are accessible for a few researchers only. To fill the gap of missing SDN enabled WSN-Testbeds we aim to design and implement a WSN-Testbed that allows a broad number of researches to test their SDN-based networking applications.

In our laboratory we have a Wireless Sensor Network Testbed (WSN-Testbed) to evaluate WSN applications. This WSN-Testbed is managed by the Testbed Management Architecture for Wireless Sensor Networks (TARWIS) (Hurni et al., 2011) that provides the most crucial management and scheduling functionalities for WSN testbeds, independent from the testbed architecture and the sensor node's operating systems. TARWIS provides rich types of network management functions, such as resource reservation features, support for reprogramming and reconfiguration of the nodes, provisions to debug and remotely reset sensor nodes in case of node failures, as well as a solution for collecting and storing experimental data.

With TARWIS as foundation we aim to create a SDN enabled WSN-Testbed that provides all the management features of TARWIS and the core components of a SDN managed network to researchers. By designing and implementing this SDN enabled WSN-Testbed we want to show the benefits of SDN in WSNs.

1.2 Objectives

The goal of this thesis is to extend the TARWIS testbed to enable rapid prototyping of SDN-based WSN network applications and routing protocols. Therefore, we formulate the following goals:

- Design a SDN-based testbed for rapid prototyping of SDN-based WSN network applications and routing protocols
- Design of a SDN-based routing protocol that uses global network information to improve the energy efficiency of the WSN
- Use the SDN-based testbed to evaluate the newly designed routing protocol
- Compare this SDN-enabled WSN routing protocol evaluation results against the one form the typical WSN scenario, where all the routing decisions are made in a fully distributed manner.

1.3 Research Questions

This master thesis tries to solve the following research questions:

- How to build a testbed for SDN-based WSN network applications?
- How can SDN-based routing protocols be designed and evaluated with this newly created testbed?
- What are the benefits that the SDN-based approach, to separate control and data plane, provides to WSNs?
- Is there an advantage of SDN-based routing protocols over distributed routing protocol with respective to energy efficiency, traffic congestion, and packet loss?

1.4 Thesis Contribution

The contributions of this master thesis can be summarized as follows:

- Design of SDNWisebed, a Software-Defined Wireless Sensor Network Testbed, an extension of TARWIS by SDN-Wise, a SDN framework for WSNs.
- Design of DTARP a Dynamic Traffic Aware Routing Protocol for Wireless Sensor Networks.
- Evaluation of SDN-based routing protocols against protocols where all the routing decisions are made in a fully distributed manner.

- Extensions of the SDN-Wise framework, that enables SDN in WSNs, to enable dynamic routing algorithms
- Improvement of the control packet routing of SDN-Wise

1.5 Thesis Structure

Within this thesis we have created several different contributions. In this Chapter 1 a brief overview over the field of this thesis is given and the motivation is sketched. This chapter also covers the objectives, the research questions and the contributions of this thesis. Finally, this section of that chapter provides an overview of the structure of the thesis and illustrates where the different contributions are explained.

To get a broader understanding of the field a literature study on the related topics was made. The results of the literature study are summarized in Chapter 2.

In Chapter 3 SDNWisebed a testbed for software-defined sensor networks is proposed and the implementation of this testbed is explained. SDNWisebed is a testbed that allows rapid prototyping of SDN-based WSN applications. The original implementation of SDN-Wise was not designed for dynamic routing algorithms and had some issues in the large real-world WSN of TARWIS and consequently, some improvements of the SDN-Wise framework were needed.

Chapter 3 explains how we have extended SDN-Wise to meet the requirements of SDNWisebed, dynamic routing, and large scale real-world deployments. With the dynamic routing extension of SDN-Wise it is possible to create dynamic routing algorithms.

In Chapter 5 the Dynamic Traffic Aware Routing Protocol for Wireless Sensor Networks (DTARP) is proposed. This protocol uses the dynamic traffic information of the WSN to find paths that reduce the risk of congestion and optimize the power consumption of the network.

In Chapter 6 experiments that have been done during this thesis are explained. The experiments show the advantage of SDN-based routing in Wireless Sensor Networks over the distributed routing protocol, highlight the advantage of a DTARP and explain the topology control of SDNWisebed.

The results of this experiments and evaluation are discussed in Chapter 7 and the conclusions are drawn. In Chapter 7 also a brief outlook for future work is given.

Chapter 2

Background and Literature

Wireless Sensor Networks (WSNs), Internet of Things (IoT) and Software-Defined Networking (SDN) have been a subject of broad interest in the research community during the past few years. In these years a lot of research has been done and a lot of results have been presented. Before starting the work in the intersection of these fields it is inevitable to do a literature and background study. In this chapter the work related to this thesis is discussed and the results of literature study about used frameworks and concepts are presented. Thus, in this chapter the common knowledge and the work of others is summarized.

2.1 Related Work

Many WSN-Testbeds have been built at universities and in the industry all over the world. K. El-Darymli et al. (2012) wrote a survey about these WSN-Testbeds and created the tools to classify and evaluate WSN-Testbeds. Even if many WSN-Testbeds exist, only a few are designed to support rapid SDN networking application deployment and testing.

Buratti et al. used the EuWIn as a WSN-Testbed for SDN measurements (Buratti et al., 2016). In their work, they used 20 sensor nodes to measure the performance of SDWN a SDN solution for WSNs (Buratti et al., 2016). To enable SDN they have added a controller that can be positioned anywhere in the network and manages the network (Buratti et al., 2016). The EuWIn WSN-Testbed provides up to 100 network nodes (50 equipped with sensors) with fixed positions and uses over the air programming to deploy the firmware (Abrignani et al., 2013). Thanks to the fixed positions and the capability to run the experiment at a defined schedule, it is possible to perform repeatable experiments.

A small real-world testbed for SDN-Wise has also been proposed by Galluccio et al. (2015). This testbed consists of five sensor nodes and a sink node deployed in a laboratory. EMB-Z2530PA based sensor nodes with a IEEE 802.15.4 wireless module have been used. The WISE-Visor and the controller of this testbed deployment was running on a single desktop computer.

Another testbed for Software Defined Networking in Wireless Sensor Networks was introduced in the Master Thesis of M. Beyene (2017). M. Beyene uses the SDN-Wise framework together with Z1 sensor nodes to enable SDN on a WSN. The testbed of Beyene consists of seven sensor nodes with no fixed locations. Therefore, he was able to test the SDN-Wise framework with different topologies. This testbed was used to evaluate SDN with different topologies and measures the convergence time, packet delay, packet loss and the control message overhead for the different topologies. This work defines the convergence time as the time needed until the flows are installed and the first packets reach their target. A comparison with protocols that use a distributed routing approach has not been done in his work.

This study of the related work showed that some testbeds to test the SDN paradigm in wireless sensor networks exist, but none of them was designed as a Multi-User Experimental Testbed (WSN-MXT). The investigated testbeds are mainly designed to test a single application and miss the management features that allow rapid prototyping of various SDN-based applications. The literature research also showed, that a benchmark test between common routing protocols and SDN-based protocols in Wireless Sensor Networks is missing or has only been executed on small testbeds.

2.2 Internet of Things

Initially, the information accessible through the Internet was mainly added by people. Thus, the computers and the Internet were mainly dependent on information added by humans. But in the past two decades this changed drastically when more and more smart devices and infrastructure, able to interact with the physical environment, were added to the Internet. These new devices started to generate and consume data independently of human interaction and, therefore, the Internet is not only about human-created information anymore but also about things (Ashton, 2009).

2.3 Wireless Sensor Networks (WSNs)

To measure physical parameters sensors are needed. When a physical parameter of a given area should be observed, often a single point of measurement is not sufficient. Enough sensor nodes need to be deployed in the area of interest to ensure the required spatial resolution. When many sensor nodes are deployed, wired networking becomes tedious and a wireless solution if often chosen. The networks that connect the sensor nodes are called Wireless Sensor Networks (WSNs). Even though WSNs can have very different properties they also have a lot in common. The sensor nodes are usually very limited in their resources. Especially the energy resources, the computational and storage capacity, signal range and traffic throughput are very constrained. Due to the limited signal range, WSNs are often used as ad-hoc networks.

Wireless Sensor Networks have been a topic of broad interest in research for over the past two decades and many very well optimized routing algorithms have been developed to route traffic from the sensor nodes to sink nodes where all the data is collected and provided to the user. Wireless Sensor Networks used to be Intranets of Things but with the trend of Internet of Things research has been done to develop those Intranets of Things to the Internet of Things. To connect the WSNs to the IoT, protocols like RPL and LoRaWAN have been developed. RPL builds a routing structure over the ad-hoc network of the WSN and each sensor is then connected to the Internet via a sink node (Winter et al., 2012). When LoRaWAN is used the smart devices are directly connected to base stations over a long range wireless connection (LoRa Alliance, 2015).

2.4 Wireless Sensor Network Testbed (WSN-Testbed)

A Wireless Sensor Network Testbed (WSN-Testbed) is an experimental environment for the testing of routing algorithms and network applications. The WSN-Testbed allows researchers and developers to deploy their projects to a realistic testing environment to get more precise testing results than by using simulators (El-Darymli and Ahmed, 2012). An example of a WSN-Testbed is TARWIS.

2.4.1 TARWIS

Deploying, testing, validation, and evaluation WSN applications can be a tedious task. Usually, the firmware needs to be installed on every sensor node and then the nodes need to be placed on their location. Already this makes the collection and distribution of the sensor nodes in larger WSNs a laborious work, but the evaluation is even so complex that it is not feasible without an assisting system. Without a system that synchronously collects the logs of all sensor nodes, the evaluation of sensor node logs is cumbersome due to the barely matched timestamps.

The Testbed Management Architecture for Wireless Sensor Networks (TARWIS) solves the deployment and log file analysis issues by connecting all sensor nodes of a WSN-Testbed over a serial cable to a single server (Hurni et al., 2011). TARWIS uses this serial interface to reset and flash the sensor nodes. Additionally, all messages sent by sensor nodes to the serial interface are logged in a single logfile on the the TARWIS service. This logfile provides a valuable tool for evaluating WSN applications. Further, TARWIS provides features to manage and schedule multiple experiments of various researchers.

2.5 Operating Systems

Sensors of WSNs mostly have severely constrained resources and therefore, a very lightweight firmware is needed. Contiki OS is a lightweight operating system that supports individual programs and services. Contiki is built around an event-driven kernel but provides optional preemptive multithreading that can be applied to individual processes (Dunkels, Gronvall, and Voigt, 2004). Contiki OS supports rapid developing of individual network applications.

2.6 Software-Defined Networking

Software-Defined Networking (SDN) is a networking paradigm that separates the control plan from the data plane (Kreutz et al., 2015). This separation of the control and the data plane reduces the complexity of the network configuration and the management. The data plane consists of network nodes that have some basic switching functions and the control plane contains the controllers. The network nodes report local information to the controller and the controller uses this information for routing or other networking applications. These routing decisions are installed in form of flow rules in the flow tables of the network nodes. The network nodes forward traffic according to the flow rules in their flow table. Therefore, the network nodes do not need to do any complex decision and the hardware can be optimized for fast forwarding. Figure 2.1 shows the SDN layer structure (Open Networking Foundation, 2012).

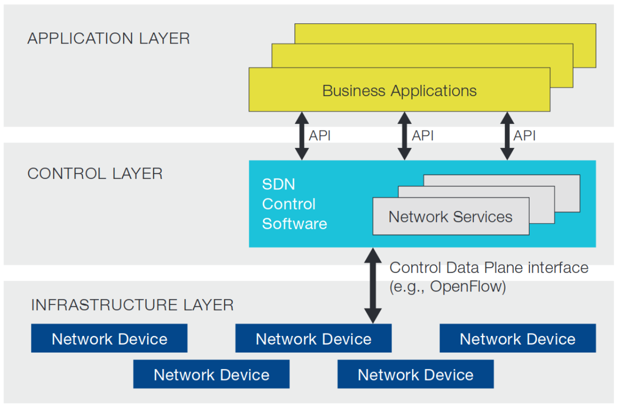


FIGURE 2.1: Software-Defined Networking (Open Networking Foundation, 2012)

With the centralization of the logic in the control plane, the management of the network has been simplified significantly. In legacy networking it was necessary to configure each switch manually, and when hardware of different manufacturers was used, even with different setups. To configure the network nodes of a SDN managed network, each node is connected to the controller and configured via a standard interface. Consequently, it is sufficient to configure the network on the controller. Another advantage of SDN is the global network view of the controller. With this information the controller is able to make much better routing decisions than a switch that has only limited information of the topology. Software-Defined Networking also enables traffic engineering to manipulate traffic flows in a more sophisticated manner.

2.7 SDN-Wise

The main goal of this thesis was to create a Testbed for SDN-based Wireless Sensor Network applications. Therefore, a SDN solution for WSNs was needed. During the past few years several SDN solutions for WSNs have been proposed and some of those solutions are already implemented (Kobo, Abu-Mahfouz, and Hancke, 2017). Due to its available and clean source code, the compatibility with the WSN in our laboratory and the advanced state of implementation we used SDN-Wise (Galluccio et al., 2015b) in this thesis. 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 deployment of novel applications, and the experimentation of new networking solutions (*SDN-Wise*).

2.7.1 Protocol Architecture

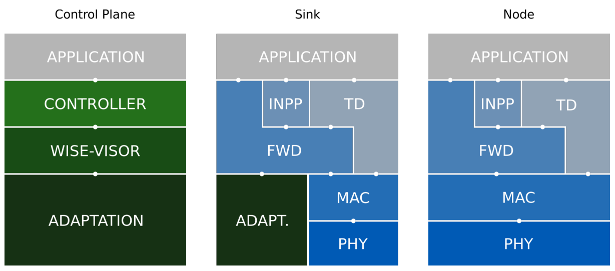


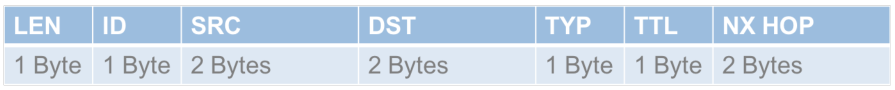
FIGURE 2.2: SDN-Wise architecture (*SDN-Wise*)

Figure 2.2 shows the protocol architecture of SDN-Wise (Galluccio et al., 2015b). The architecture consists of a control plane a sink and node elements. The control plane is made up of the adaptation layer, which transforms the messages received from the sink to a format that can be handled by the WISE-Visor. The WISE-Visor distributes the traffic of different logical networks to the corresponding controllers. The control layer consists of controllers that handle the controller requests. The sink and node network element (shown in Figure 2.2) consist of the IEEE 802.15.4 physical and MAC layer. On top of the MAC layer is the Forwarding (FWD) layer that handles incoming packets according to the WISE flow table. The Topology Discovery (TD) component is used to discover the local neighborhood of a sensor node and reports this information to the controller. With the In-Network Packet Processing (INPP) unit it is possible to run operations like data aggregation or other in-network processing functions on top of the Forwarding layer (FWD). In addition to these components the sink also has an adaptation unit to pass packets to the control plane (*SDN-Wise*).

2.7.2 Packet Formats

The packets defined by SDN-Wise build an important component to create new network applications. Consequently, it is essential to understand the packet definitions. In this section the packet definitions of SDN-Wise are briefly summarized (*SDN-Wise*).

Each packet contains a header shown in Figure 2.3. The header is used to describe the packet with its length (LEN), network id (ID), source node address (SRC), destination node address (DST) and the type (TYP) of the packet. For each hop the packet was sent over the Time To Live (TTL) countdown is decreased and when it reaches zero the packet is dropped. NX Hop contains the address of the next hop.



LEN	ID	SRC	DST	TYP	TTL	NX HOP
1 Byte	1 Byte	2 Bytes	2 Bytes	1 Byte	1 Byte	2 Bytes

FIGURE 2.3: Header of packets

The data packet shown in Figure 2.4 with TYP = 0 is the packet that can be used to transmit application specific data. It consists of the header and the payload. The payload can be any format as long as the application is able to encode it. The maximum size of the payload for the TelosB mote integration is 106 bytes. For larger data packets, the message needs to be split into 106 byte segments.

Header	Payload
10 Bytes	n Bytes

FIGURE 2.4: Data packet

The beacon packet shown in Figure 2.5 is used to discover the neighbor sensor nodes and the next hop towards the closest sink node. It contains a header with TYP = 1, the distance from the sending node to the closest sink node, and the battery level of the sending node.

Header	Distance	Battery
10 Bytes	1 Byte	1 Byte

FIGURE 2.5: Beacon packet

The report packet with TYP = 2 shown in Figure 2.6 is used to report local neighborhood information to the controller. This packet consists of the header, the distance to the next sink node, the battery level of the sending node, the number of neighbors in the neighbor table of the sending node and the neighbor addresses associated with the Radio Signal Strength Indicator (RSSI) level of the link between the node and the neighbor. The RSSI represents the signal strength of a received packet and is a value between 0 and 255.

Header	Distance	Battery	#Neighbors	Address 1	RSSI 1	…	Address n	RSSI n
10 Bytes	1 Byte	1 Byte	1 Byte	2 Bytes	1 Byte	…	2 Bytes	1 Byte

FIGURE 2.6: Report packet

The request packet with TYP = 3 shown in Figure 2.7 is sent to the controller to request a flow entry, when a sensor node has no matching flow entry to handle a packet. The request packet consists of the header, the request id and the packet that does not have a matching flow entry. It might happen that the request packet with the additional header information exceeds the payload limit and the unmatched packet does not fit into one request. When the request does not fit in one packet, then the unmatched packet is split into two parts. The part byte denotes the index of the contained part and the total contains the number of parts in total.

Header	Id	Part	Total	Unmatched Packet
10 Bytes	1 Byte	1 Byte	1 Byte	n Bytes

FIGURE 2.7: Request packet

When the controller has processed a request packet, it sends a response packet shown in Figure 2.8 with TYP = 4 to the sensor node that has sent the request. This response packet contains the header and the flow rules to install on the sensor node.

Header	Rule
10 Bytes	n Bytes

FIGURE 2.8: Response packet

To install a complete path in the network by sending request and response packets can be very inefficient as each sensor node on the path needs to request the flow entries from the controller. To avoid these multiple requests when a path is installed the OpenPath packet can be used. The OpenPath packet shown in Figure 2.9 with TYP = 5 is used to install a route between two sensor nodes. When the path has been installed each sensor node on the path is able to send packets to the source and the destination of this path. The OpenPath packet consists of the header, window size, the windows for the flow entries and a list of addresses that need to be traversed to send a packet from the source to the destination. To install the path, an OpenPath packet is sent to the requesting sensor node. When the requesting sensor node receives the OpenPath packet it installs the first window and the first address as the next hop. Then, the OpenPath packet is forwarded to the next node on the path. Therefore, only one request needs to be sent to the controller.

Header	Windows Size	Window 1	...	Window n	Address 1	...	Address k
10 Bytes	1 Byte	5 Bytes		5 Bytes	2 Bytes		2 Bytes

FIGURE 2.9: OpenPath packet

The config packet shown in Figure 2.10 with TYP = 6 is used to set configuration options on a sensor node. The packet consists of the header, the id of the configuration that should be changed, and an optional parameter. This parameter could be the value that should be set.

Header	ConfigId	Params (Optional)
10 Bytes	1 Byte	n Bytes

FIGURE 2.10: Config packet

Each sink node needs to send a RegProxy packet with TYP = 7 shown in Figure 2.11 to the controller to announce its existence. This packet consists of the header, the id of the sink, the MAC address of the sink, the physical port, the sink is connected to, the IP address of the sink and the TCP port of the sink. With this information the controller is able to send packets to the sensor nodes reachable over this sink.

Header	DPID	MAC	Port	IP	TCP
10 Bytes	8 Bytes	6 Bytes	8 Bytes	4 Bytes	1 Byte

FIGURE 2.11: RegProxy packet

2.7.3 Sensor Node Firmware

The SDN-Wise framework also contains a sensor node firmware that can be used to bootstrap SDN-based network applications. This firmware is based on the Contiki OS and was introduced in (Dio et al., 2016; Galluccio et al., 2015a; Galluccio et al., 2015b). Thanks to the use of Contiki OS, the SDN-Wise Contiki firmware can be installed on a wide set of sensor nodes. SDN-Wise Contiki implements most of the functions of the SDN-Wise framework and, therefore, it is possible to use it for application development.

2.7.4 Controller

The SDN-Wise framework includes two controller options. The first is SDN-Wise Java, a Java application that contains the adaptation layer, the flow visor and a simple controller (Dio et al., 2016; Galluccio et al., 2015a; Galluccio et al., 2015b). This controller uses the Radio Signal Strength Indicator (RSSI) value as weight for shortest path routing. The SDN-Wise Java controller is not so flexible and easy to extend and, therefore, it does not support rapid prototyping of network applications.

For rapid prototyping the Open Networking Operating System (ONOS) in combination with the SDN-Wise controller plugin is much more suitable (Berde et al., 2014; Anadiotis et al., 2018; Anadiotis et al., 2015). ONOS is written in Java and the source is publicly available. ONOS consists of a very well layered architecture with north and southbound interfaces (Berde et al., 2014).

To use ONOS as controller SDN-Wise Java is needed as Wise Visor. SDN-Wise Java is configured to forward the incoming messages from the sink node towards the ONOS controller.

For ONOS a SDN-Wise plugin that exists (Anadiotis et al., 2018; Anadiotis et al., 2015). This plugin provides a Flow Rule Service, a Topology Service, Sensor Node Service and Packet Service that can be used by the developers in their own network applications. All these services enable rapid prototyping of network applications and SDN-based routing protocols.

2.7.5 Packet Routing and Forwarding

The forwarding of packets between a source and a destination is the most important task in networks. That a packet can be sent from source to destination a routing protocol that routes the packet through the network is needed. In SDN-Wise two types of routing protocols are needed.

First, a SDN-based routing algorithm is used to route node to node and sink to node packets through the network. For this SDN-based packet routing the standard implementation of SDN-Wise uses the Dijkstra shortest path algorithm with the Radio Signal Strength Indicator (RSSI) as cost function for routing (Dio et al., 2016; Galluccio et al., 2015a; Galluccio et al., 2015b). This standard implementation also uses a reactive routing approach to install routes in the network. When a sensor node processes a packet it matches the packet with all flow entries of the flow table and when a flow entry matches, this packet is handled accordingly to this flow entry. When a packet does not match with any flow entry a request packet is sent to the controller. The controller calculates the path from the requesting sensor node to the destination of the packet and sends a response or an OpenPath packet to the requesting node. A response only contains the next hop of the path and the OpenPath packet installs the route from the requesting node to the destination. Therefore, with the OpenPath packet it is enough to send one request to the controller and with the response packet each node on the path needs to request a flow entry.

As second routing protocol, a tree based routing is used to send control packets from the sensor nodes to the controller. This control packets are needed to maintain the SDN network and, therefore, also need to find their destination when the SDN network is not established yet. This control packet routing tree is built from the sink node by sending a broadcast with the distance to the sink through the network (Dio et al., 2016; Galluccio et al., 2015a; Galluccio et al., 2015b).

2.8 Routing Protocols

Many routing protocols for WSNs already exist. In this Section three WSN routing protocols that are relevant for this thesis are discussed. This protocols will be used to evaluate the benefits of the SDN approach to separate the control plane form the data plane. First, distributed protocols Flooding and RPL are summarized. Afterwards, the standard routing protocol equipped with the SDN-Wise framework, SDN-Wise (RSSI), is explained.

2.8.1 Flooding

Flooding is the simplest protocol to disseminate information in a network. Flooding is based on broadcast messages that are used to send information to all neighbors. The neighbors receiving this information retransmit it as fast as possible to all their neighbors. Flooding is a special case of a gossip protocol, which sends the information to all direct neighbors and without a delay before retransmission. With the general gossip protocol, the packets are retransmitted to a subset of the neighbors and with a given delay before retransmission. Gossip protocols are very robust, as the information is spread over multiple paths and, therefore, the nodes receive the information multiple times (Jelasity, 2013).

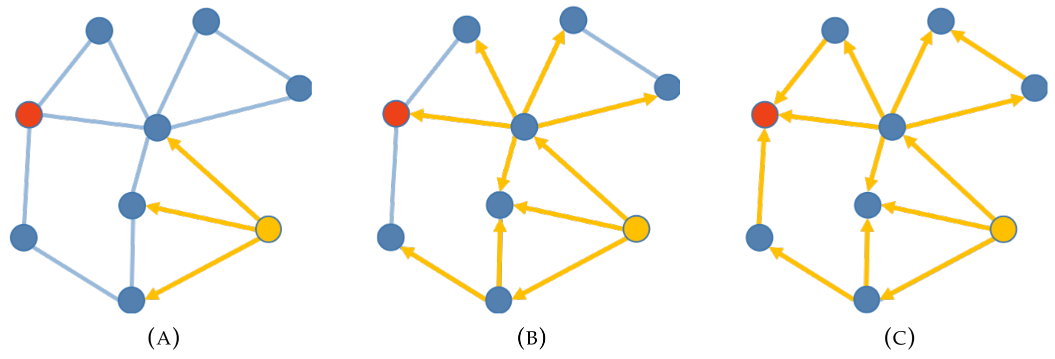


FIGURE 2.12: Flooding Information Dissemination

Figure 2.12 shows an example of information dissemination with flooding. In (A) the source node broadcasts the message to all its neighbors. The neighbors that received the message rebroadcast it to all their neighbors (B). Now, the message already reached its destination, but the other nodes do not have this information and rebroadcast the packet again (C) to their neighbors and repeat this rebroadcasting until the Time to Live TTL of the packet reaches zero. In a more advanced version of the Flooding protocol, each node keeps track of the packet ids it has sent and only retransmits a message when it does not have already broadcasted a message with this id. With this improvement, the overhead of the protocol can be reduced.

2.8.2 Routing Protocol for Low power and Lossy Networks (RPL)

Most Wireless Sensor Networks are used to gather environmental parameters over an area of interest. To collect this information sensor nodes are distributed over the area and the sensor sample physical parameters. These sensor readings are then sent to designated sink nodes that collect all information and expose this data to other services. For collecting the data, all sensor nodes send data packets towards the sink nodes. In ad-hoc sensor networks this node-to-sink flows include multiple hops.

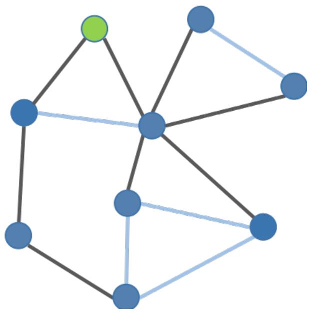

(A)

FIGURE 2.13: RPL Overlay Tree

The Routing Protocol for Low power and Lossy Networks (RPL) was designed and optimized for node to sink flows in WSNs. RPL builds and maintains a Destination Oriented Directed Acyclic Graph (DODAG) to route the traffic form the nodes to the sink (Winter et al., 2012). To build the DODAG the sink node issues a new tree version and announces this tree version, along with a depth of zero in a broadcast message. When a node receives a tree version package and the version of the packet is newer than the own version and the depth announced in the packed is lower than the own depth, then it updates its own depth to (packet depth + 1) and rebroadcasts the tree version packet with the updated depth (Winter et al., 2012). Figure 2.13 shows an example of a DODAG. With this DODAG the transmission of data packets from the nodes to the sink is as simple as forwarding every packet to the parent node. RPL is very efficient for node-to-sink communication. With RPL the DODAG is updated continuously and therefore, the protocol reacts dynamically to topology changes.

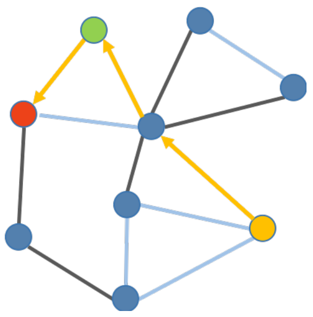


FIGURE 2.14: RPL Information Dissemination

With the sensor nodes becoming smarter, the requirement for sink-to-node and node-to-node flows is rising. RPL also supports node-to-node communication, but as it is highly optimized for node-to-sink communication, it has a huge drawback in efficiency. To enable node-to-node communication, each node sends a list with its child nodes towards the source node. In the non-storing mode this child information is stored on the sink node only. With this additional information downwards and, consequently, also node-to-node routing is possible. To send a packet form a source node to a destination node the packet is sent towards the sink node and from there, the packet is sent down the DODAG to the destination node. Figure 2.14 shows how a packet is sent from the source to its destination. First, the packet is sent along the DODAG towards the sink node. Then the packet is sent to the root node of the child tree that contains the destination until the destination is reached.

RPL also supports a storing mode where each node keeps track of its subtrees and routing entries (Winter et al., 2012). In this mode each node maintains a list of the node that belong to its subtrees. When an intermediate node receives a packet, that is addressed to a node contained in one of its subtrees, then the intermediate node is forwarding the packet downwards this subtree instead of sending it to its parent. Consequently, not all packets are sent over the root node.

2.8.3 SDN-Wise (RSSI)

Using the global WSN information of the controller, SDN-based routing algorithms have much more potential for optimal routing than distributed algorithms that must use local information. Using SDN-Wise the packets can be sent along each desired path in the network. To make the routing decisions, the default SDN-Wise algorithm uses the Radio Signal Strength Indicator (RSSI) of the links as cost function to calculate the shortest path (Dio et al., 2016; Galluccio et al., 2015a; Galluccio et al., 2015b).

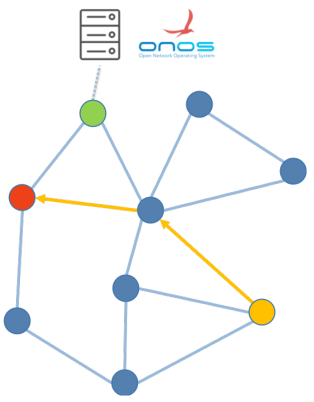


FIGURE 2.15: SDNWise RSSI Information Dissemination

Figure 2.15 shows how a packet is sent from its source to its destination. Each node, that does not have a flow table entry for the destination sends a request to the controller. The controller calculates the shortest path according to the RSSI metric and returns a response towards the requesting node. When all flow entries of the path are installed the data packets are sent over the shortest path.

Chapter 3

SDNWisebed

This Chapter describes how we extended TARWIS with the SDN-Wise framework to create SDNWisebed, a WSN-Testbed for Software Defined Networking. SDNWisebed is a Multi-User Experimental Testbed (WSN-MXT). A WSN-MXT allows multiple researchers to deploy and test their projects (El-Darymli and Ahmed, 2012). Therefore, SDNWisebed can be used for rapid research on various SDN-based network applications by different researchers.

In this Chapter, first the design of SDNWisebed is introduced and afterwards in Section 3.2 the implementation of SDNWisebed is discussed. To implement SDNWisebed we needed to extend the SDN-Wise framework. These extensions of the SDN-Wise framework are described in Section 3.2.3.

3.1 Testbed Design

The main goal of a WSN-Testbed is to provide researchers with an environment where they can easily and continuously test their projects. Therefore, easy to use tools for experimentation management, firmware deployment, resource sharing and allocation, topology control, data collection, and experiment execution are needed.

Am SDN-based WSN network application normally consists of the sensor node firmware and a controller application. A SDN managed sensor network additionally needs a SDN controller and a method to deploy network applications to the controller. Furthermore, a border router is needed to exchange information between the sensor nodes and the controller.

In Figure 3.1 the architecture of SDNWisebed is shown. The architecture is split into three layers. The top layer is the control plane, known from the SDN paradigm. In the control plane the SDN controllers can be found. These controllers are used to gather network information and to run network applications. In the middle layer the sensor nodes and border routers can be found. This layer is the data plane, known from the SDN paradigm. In this layer, topology information is gathered and sent to the controller and packets are forwarded accordingly to the controller directives. The third layer is used to manage the sensor nodes. It provides all the functionalities needed to deploy projects of numerous researchers to the sensor nodes, the resource sharing and allocation, topology control, data collection, management, and experiment execution. Providing all these services, this layer can be seen as a Metal as a Service (MaaS) for sensor nodes.

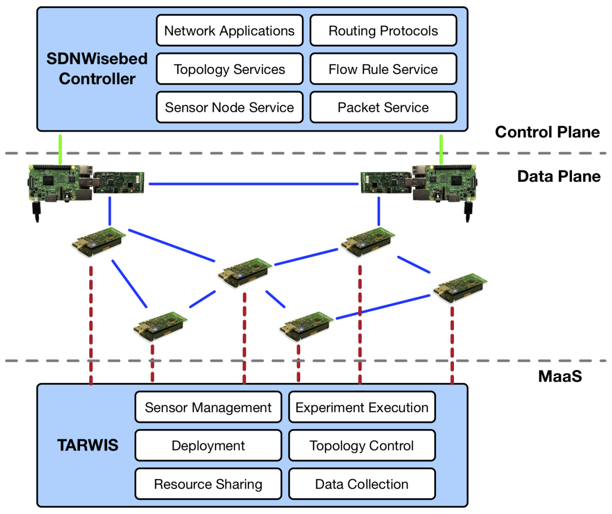


FIGURE 3.1: The architecture of SDNWisebed

In the following more detailed descriptions of the most important components of SDNWisebed can be found.

3.1.1 SDN Network Applications

The SDN network application is the piece of software that is tested in SDNWisebed. A SDN network application for WSNs is made of two components. The first component is the firmware on the nodes. This firmware needs to implement the application and provide the controller with the information it needs for further processing. The second component is the controller application plugin. This application plugin has access to the complete information that the controller has collected about the sensor network. The controller application can access sensor node information from the Sensor Node Service or handle incoming packets with the Packet Service. Using the Topology Service, the shortest paths between two nodes can be calculated and the calculated routes can then be installed on the sensor nodes by using the Flow Route service.

When experiments are executed in a real testbed, it is important that the experiments are repeatable, and the noise is as low as possible. Therefore, we focused on using components in SDNWisebed that allow repetitions of experiments with the same parameters, like the network topology, the time of execution etc.

To enable rapid prototyping, a fast research-feedback-loop is mandatory. Consequently, the controller and the operating system of the sensor nodes should provide libraries so that researcher can focus on the development of the network application of interest and the quick installation and re-installation of applications is compulsory.

3.1.2 SDN Controller

In a network following the SDN paradigm the SDN controller (controller) builds the core component. The controller gathers information about the network topology, does routing decisions and runs network applications.

The controller used in SDNWisebed can be extended with network applications. The controller provides services that serve all information it has gathered from the network. With this information, the researcher can build its own network applications. The controller also provides services to change the routes in the network. With the ability to change routes in runtime it is possible to apply traffic engineering to the WSN.

To support the researcher in the process of tweaking their algorithms and networking applications fast redeployment methods are crucial. The controller used by SDNWisebed allows to activate and deactivate addons in runtime, and, therefore, the need for a fast development cycle is fulfilled. Furthermore, the controller allows to write log files about routing decisions, traffic statistics, sensor node states etc. These logfiles can later be analyzed to evaluate the network application under test.

3.1.3 Sensor Nodes

The sensor nodes of SDNWisebed fulfill two purposes. First, they need to forward packets through the WSN. In this case, they build the data plane of the SDN paradigm. Therefore, the sensor nodes match packets with their internal flow table and execute the action of a matching flow entry. When a packet does not match with any flow entry, the sensor node sends a message to the controller to request a new flow entry (Galluccio et al., 2015a). The second purpose of a sensor node is to interact with the environment. This can either be the sensing of a physical property or an action. This interaction with the environment is part of the application layer and can be tested and evaluated with SDNWisebed.

To integrate a new program on the application layer of the sensor node, the firmware can be extended by the needed functionality. This application can exchange information with other sensor nodes or the controller.

3.1.4 Border Router

In a SDN managed network, information needs to be exchanged between the control plane and the data plane. In SDNWisebed the data plane is formed by the sensor nodes and an ONOS server is used as SDN controller. Control information must be exchanged between the sensor nodes and the ONOS controller. To exchange this information between the wired controller and the wireless sensor nodes a border router is required. In SDNWisebed a sensor node in sink mode connected to a Raspberry Pi is used as border router. The Raspberry Pi is connected to the laboratory network and can send packets to the controller and the sensor network. On the Raspberry Pi the WISE-Visor of SDN-Wise Java (Galluccio et al., 2015a) is running as flow visor to manage the flow of the control packets and data packets incoming or outgoing the WSN. A flow visor is a transparent proxy between the sensor nodes and multiple controllers. The flow visor can delegate different slices of the network to specific controllers.

In a SDN managed WSN, the control packets always need to be sent to the border router, thus the sink nodes, which are part of the border routers, must handle a lot of traffic in big networks. Therefore, the sink node of a WSN is a bottleneck and when too many sensor nodes are added the it is not able to handle all traffic. Consequently, the network is not scalable with a single sink node. To leverage scalability, multiple sink nodes can be added to the WSN. The control packets are always sent to the closest sink node. In SDNWisebed it is possible to set up multiple border routers to be able to test applications that requires multiple sinks.

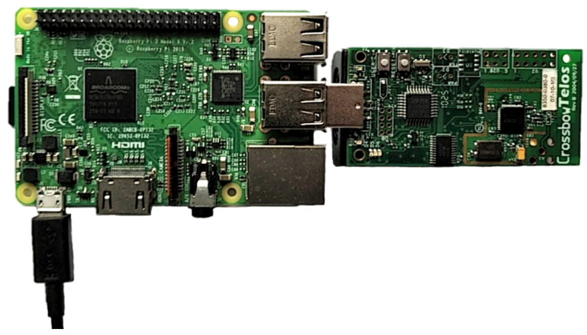


FIGURE 3.2: Border router with Raspberry Pi and TelosB

3.1.5 Topology Control

SDNWisebed consists of 40 permanently installed TelosB sensor nodes in two buildings of our institute. This permanent installation allows the execution of repeatable experiments. To use SDNWisebed with different physical network topologies, two different methods are available. First, with TARWIS, which is used in SDNWisebed, it is possible to activate only a subset of the sensor nodes (Hurni et al., 2011). Therefore, a lot of different and repeatable topologies can be deployed and tested by only activating the necessary nodes.

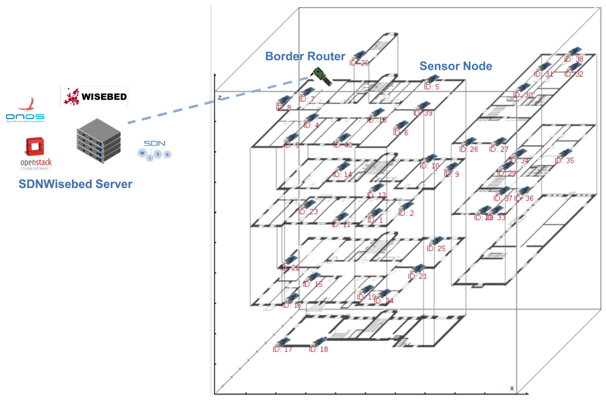


FIGURE 3.3: Deployment of the sensor nodes of TARWIS

The second method to change the topology, is to change the transmission power of the sensor nodes. The change of transmission power also changes the range of the signal and, therefore, the topology of the network. As we will show in Chapter 6, the change of transmission power has an impact on the topology. Thus, we are able to test SDN-based applications in sensor networks with different network characteristics.

The transmission power level can be changed in the Contiki OS application. To change the power level the method of Listing 3.1 can be used.

```
cc2420_set_txpower(TX_POWER_LEVEL);
```

LISTING 3.1: Change Transmission Power (Contiki OS)

3.2 Testbed Implementation

3.2.1 Software platform

Our testbed relies on available SDN technologies and open source software components. In particular, we use the SDN-Wise framework (Galluccio et al., 2015b; Galluccio et al., 2015a; Anadiotis et al., 2018; Dio et al., 2016) and TARWIS (Hurni et al., 2011).

As controller software we use the Open Network Operating System (ONOS) (Berde et al., 2014), which provides well-defined interfaces between the different layers and, therefore, enables a clear separation of networking applications and drivers. ONOS also provides an easy to use plugin system, which enables rapid application development. To enable SDN-Wise on ONOS we use the SDN-Wise plugin for ONOS (Anadiotis et al., 2018; Anadiotis et al., 2015).

On the border router we run SDN-Wise Java as flow visor to manage the packet exchange between the wired and wireless network (Galluccio et al., 2015a).

To run the TelosB motes, we use Contiki OS (Dunkels, Gronvall, and Voigt, 2004) with the SDN-Wise application (Galluccio et al., 2015b). Contiki OS and SDN-Wise gives the researcher the opportunity to focus on the development of the network application and removes the need to deal with the lower network layers.

For the management of the sensor nodes, firmware deployment, resource sharing and allocation, topology control, data collection, and experiment execution we use TARWIS (Hurni et al., 2011).

3.2.2 Hardware platform

SDNWisebed consists of 40 programmable sensor nodes based on the TelosB mote. The TelosB mote is an IEEE 802.15.4 compliant wireless sensor node equipped with temperature, humidity, and light sensors. All sensors are connected to TARWIS over the serial interface. The sensor nodes are fully controllable over TARWIS. Figure 3.3 shows the deployment of the TelosB sensor nodes in our institute (Hurni et al., 2011).

For the border router we use a Raspberry Pi 3 Model B running Raspbian 4.14. This Raspberry Pi is connected to a TelosB sensor node running in sink mode.

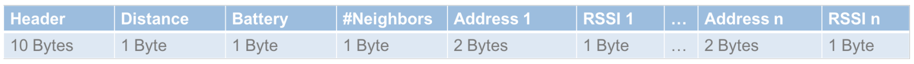
TARWIS and the controller are running in virtual machines on top of an OpenStack cluster.

3.2.3 Extensions of SDN-Wise

To create SDNWisebed and the Dynamic Traffic aware Routing Protocol DTARP, which is introduced in Chapter 5, we used the SDN-Wise framework. During the project we have extended the SDN-Wise framework by some new features and improvements. This section covers the motivation and realization of the most important changes we contributed to the SDN-Wise framework.

Report Update

Separating the control from the data plane requires the nodes of the network to report their local information to the controller. The controller requires this information to build a global view of the network and for routing decisions. In SDN-Wise periodically sent report packets are used to update the controller about local changes. In the original version of SDN-Wise the battery level and RSSI levels to adjacent neighbors are reported to the controller. Figure 3.4 shows the format of the original report packet. One of our goals was to create a routing protocol that dynamically reacts to

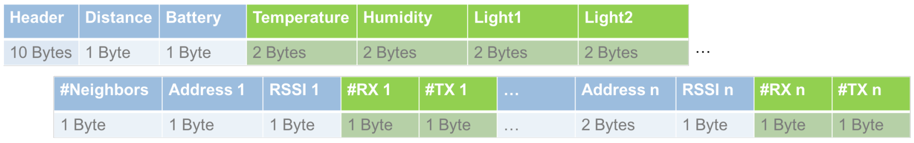


Header	Distance	Battery	#Neighbors	Address 1	RSSI 1	...	Address n	RSSI n
10 Bytes	1 Byte	1 Byte	1 Byte	2 Bytes	1 Byte	...	2 Bytes	1 Byte

FIGURE 3.4: Original Report Packet of SDN-Wise

local changes. Therefore, more information about the sensor node and its environment were required. To fulfill this requirement, we extended the report packet by sensor readings and traffic statistics. Figure 3.5 shows the structure of the updated report packet. We have added temperature, light, and humidity values of the sensor nodes to the report packet. This additional information allows more advanced routing algorithms. For example, the light information can be used by a routing protocol to increase the routing cost over sensor nodes which are not exposed to light. Such a routing protocol might be advantageous when the sensor nodes are powered by solar cells. The traffic statistics we have added to the report packets, can be used to determine the radio module workload of sensor nodes and, therefore, can be used to avoid congestion on sensor nodes.

In our updated version of the report packet, for each adjacent sensor node two bytes are reserved for traffic statistics. Those two bytes can be used in two modes. In the first mode the two bytes are combined, and for each incoming and outgoing packet, the value of this combined byte is incremented. The advantage of this mode is that many more packets can be counted until the variable maximum is reached and the packet does not need to be sent that often to the controller. In the second mode, one byte is used for the ingress traffic from the adjacent node and the other for the egress. In this mode the report needs to be sent more frequently, as the counting variable is only one byte. The advantage of this mode is the more fine-grained information about the traffic patterns in the WSN that is sent to the controller. With these updated



Header	Distance	Battery	Temperature	Humidity	Light1	Light2	
10 Bytes	1 Byte	1 Byte	2 Bytes	2 Bytes	2 Bytes	2 Bytes	...

#Neighbors	Address 1	RSSI 1	#RX 1	#TX 1	...	Address n	RSSI n	#RX n	#TX n
1 Byte	1 Byte	1 Byte	1 Byte	1 Byte	...	2 Bytes	1 Byte	1 Byte	1 Byte

FIGURE 3.5: Extended Report Packet of SDN-Wise

report packets, the controller has a more detailed view about the network and it is possible to implement more sophisticated network applications. The change of the protocol was also a proof of concept that it is possible to adapt the SDN-Wise framework to application specific needs. This change required the update of the sensor node firmware and SDN-Wise Java.

Flow Rule Updates

In a dynamic routing protocol the routes change over time. Therefore, old paths need to be removed and new paths must be installed. To remove a path from the network each flow entry that belongs to this path needs to be deleted.

The original SDN-Wise implementation does only support the removal of flow entries from the flow table by config packets. The config packet to remove a flow entry requires the position of the flow entry in the flow table. The information where the flow entry is stored in the flow table is determined by the sensor node when the flow entry is installed and consequently not known by the controller.

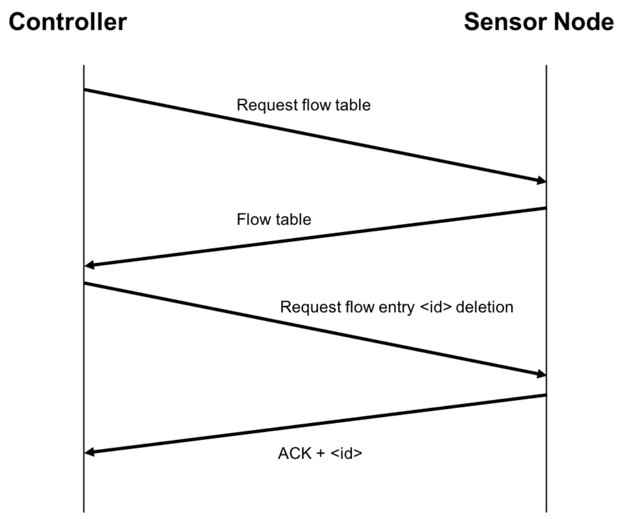


FIGURE 3.6: Deletion of a flow entry from the flow table

Figure 3.6 shows the sequence diagram of a flow rule deletion initiated by the controller. To remove an entry from the flow table of a sensor node the controller needs first to request the flow table of the sensor node. The flow table in the response of this request contains all flow entries of the sensor node, and ids of these flow entries. The controller can use the id of the flow entry, of the requested flow table, to delete the flow entry on the sensor node. But therefore it needs to send another packet to the sensor node. To ensure that the flow entry was really deleted an acknowledgement is needed. This acknowledgement is mandatory, as a remaining wrong flow entry would route the traffic into the wrong direction. Consequently, at least 4 packets need to be exchanged between the controller to delete a single flow entry from one sensor node. But when a complete path needs to be removed to change the flow of the traffic this deletion must be sent to each sensor node on the path. This procedure creates a lot of overhead and is difficult to manage. Consequently, it is not feasible to change an once installed route with the original SDN-Wise framework, and consequently, dynamic routing protocols are not possible. To enable dynamic routing at least the removal of routing entries is needed, so that the routing can change over time.

One of the goals of this work is to design, implement, and test a dynamic SDN-based routing algorithm. Therefore, the ability to update flows is essential and required.

The update of the flow tables could be designed and implemented in several different ways. Either the update is managed on the controller or on the senor node. If the update is managed on the controller, the controller needs to keep track of all installed routes and it needs to send flow entry update packets to the sensor nodes when routing decisions are changed. But this also requires a flow entry identifier that is assigned by the controller to remove the need of requesting flow tables from the sensor nodes. If the update is managed on the sensor node, it is easiest done by deleting flow entries and requesting new ones. But, therefore, the sensor node must decide when an update is needed.

The simplest and also quite efficient way to implement the flow entry update function on the sensor node is to delete the flow entries when they are expected to expire. When a flow entry for the routing of a certain destination is missing, the sensor node is sending a routing request to the controller and the new and updated flow entry is installed by the controller. Several options to implement the deletion interval are available, for example it could be adaptive to the use of a flow entry or it could be static. In our implementation we used a static interval. Figure 3.7 shows a data flow diagram of the flow table update process. When the controller sends an OpenPath packet or a response to a routing request, the flow entries contained in this packet are added to the flow table of the sensor node. When the flow table drop interval has expired the flow table entries are dropped.

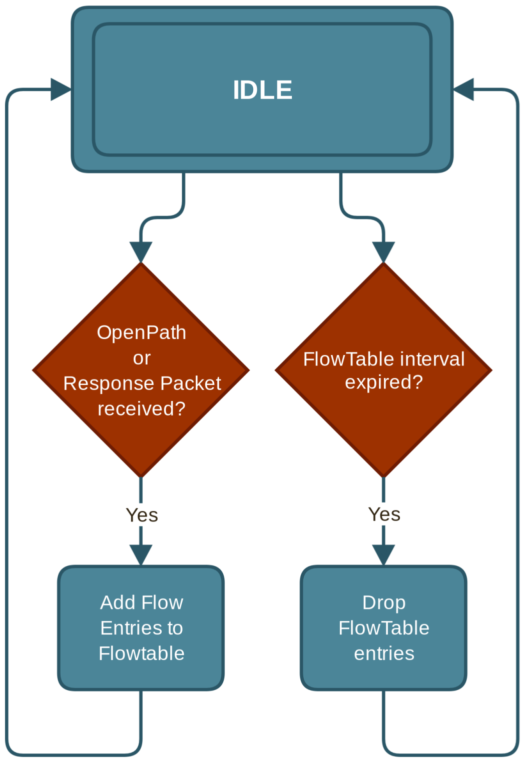

IDLE
OpenPath or Response Packet received?
FlowTable interval expired?
Yes
Yes
Add Flow Entries to Flowtable
Drop FlowTable entries

FIGURE 3.7: Flowtable Update Process

When all sensor nodes drop their flow tables at the same time, all sensor nodes have to request new flows at the same time. When all requests are sent simultaneously the sink nodes might be flooded and congested. To prevent all sensor nodes from dropping their complete flow table at the same time, we added an initial delay of random length before the periodic drop of the flow table is initiated.

Nevertheless, our current implementation is a working, but it is not well a studied solution. To improve dynamic routing protocols in Wireless Sensor Networks, the update of flow entries should be investigated, and different solutions should be compared in future work.

Control Packet Routing: Tree Versions

To route control packets from the sensor nodes through the sink node to the controller a tree-based routing is used in SDN-Wise (Dio et al., 2016; Galluccio et al., 2015a; Galluccio et al., 2015b). The tree used to route the packets through the network is built by beacon messages.

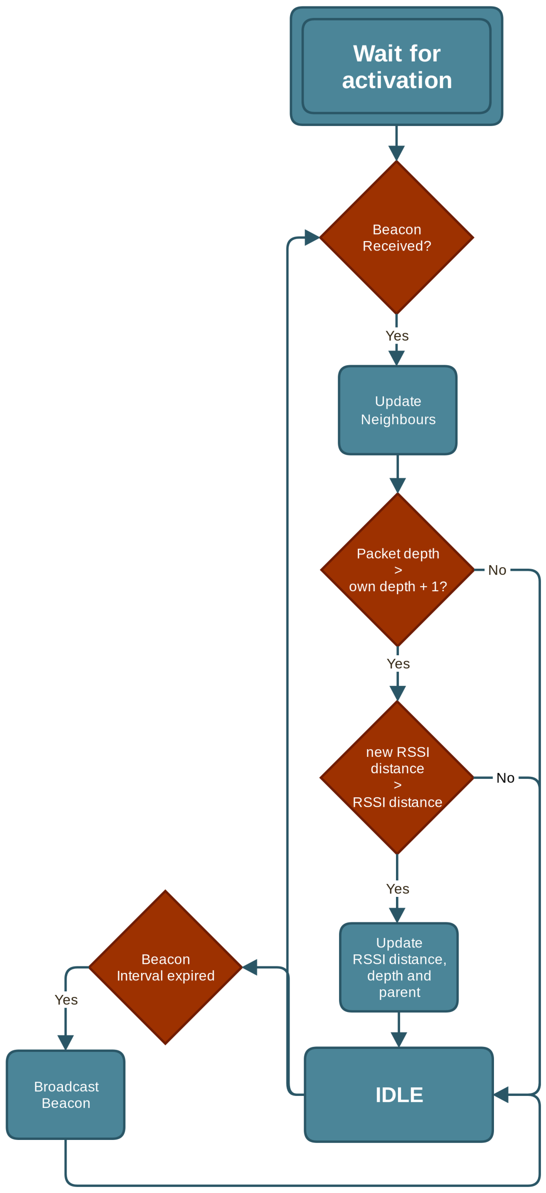


FIGURE 3.8: Handle Beacon SDN-Wise (Original)

In the original version of SDN-Wise each sensor node stores the number of hops to the next sink node and the cumulative RSSI of the links to the sink node. Initially, the local number of hops to the sink node (depth) is set to 255 and the cumulative RSSI value is set to 0. In Figure 3.8 a data flow diagram of the original SDN-Wise beacon handler is shown. When a sensor node receives a beacon message, it first updates the neighbor table with the RSSI value of the received packet and the id of the sending sensor node. When the depth information of the received beacon is smaller than depth + 1 stored on the sensor node, it is checked if the cumulative RSSI value of the beacon is greater than the locally stored cumulative RSSI value and if this is true the sink information of the node is updated. In this update the parent of the current node is set to the sender of the handled beacon, the local cumulative RSSI value is set to the corresponding value of the beacon and the depth is set to depth + 1 of the beacon. Initially, the sensor nodes are not activated and are just listening. The sensor node is activated by the first received beacon. An active sensor node is broadcasting its depth in a certain interval. The sink node is broadcasting in a certain interval from the beginning. Hence, the neighbors of the sink node are activated when they receive the first sink node beacon and then the whole sensor network is activated gradually.

This tree creation works well in small testbeds and in simulations, but it is prone to errors in a real-world environment. The major problem of the approach shown in Figure 3.8 is that it can happen that the sensor nodes update outdated information. In real-world deployments overhearing and collisions are a quite common problem and, hence, sensor nodes do not receive all beacons and it might happen that they disseminate outdated tree information. As the beacon packet does not contain a version number, the receiving node is not able to validate actuality of the tree update and, therefore, it must accept the packet. When the topology is not stable, the dissemination of outdated tree information can lead to cycles in the routing tree. When a cycle occurs in the routing graph, the network builds clusters, and the control packets cannot be sent to the controller anymore.

To overcome this problem, we changed the algorithm that builds the tree for the control packet routing. First, we decoupled the neighborhood discovery beacon from the tree building beacon. This gave us the ability to implement an RPL like tree building algorithm.

Header	Distance	Battery
10 Bytes	1 Byte	1 Byte

FIGURE 3.9: Original SDN-Wise beacon packet

Figure 3.9 shows the original beacon. It consists of header, distance to the closest sink node and battery level of the sensor. To decouple neighborhood discovery from the tree building algorithm we added a type and tree version byte to the beacon packet. Figure 3.10 shows the extended beacon. Type contains the type of the beacon. The beacon can either be of type tree = 1 or neighbor = 2. In the tree version field the version of the tree is stored.

Header	Type	Tree Version	Distance	Battery
10 Bytes	1 Byte	1 Byte	1 Byte	1 Byte

FIGURE 3.10: Extended beacon packet

Figure 3.11 shows how we have decoupled the neighbored discovery from the tree update. The major change to the original implementation of the tree building algorithm is that the sink node issues new tree versions and broadcasts a new tree beacon. This tree version is disseminated through the network by rebroadcasting the tree beacon when a node was updated. Thus, the sensor nodes do not have a fixed beacon interval and only the sink node is issuing updates. So, the algorithm used to build the tree for routing the control packets is like the algorithm used to build the DODAG in RPL (Winter et al., 2012; Schärer, Zumbrunn, and Braun, 2017).

In the new tree building algorithm shown in Figure 3.11 sensor nodes only accept a tree beacon unconditionally if the tree version of the received packet is higher than the tree version currently stored at the node. If the tree version of the received beacon is equal to the stored tree version, then the tree version is only updated if the depth to the sink node is shorter with this beacon information. If there is tie in the depth to the sink node then the tree information is only updated, if the RSSI value of the received beacon is better than the stored value. This algorithm guarantees a cycle free tree (Winter et al., 2012).

No neighborhood information of the sensor nodes is needed, for node-to-sink routing with the RPL like algorithm that is used to send control packets over the sink node to the controller. Therefore, this task can be decoupled form the tree building algorithm. Consequently, we implemented a new beacon that only contains the battery level and the id of the sending node. This neighborhood beacon is only used to announce the existence of a node to its neighbors and is sent in a certain interval. Each node that receives such a beacon is updating its neighbor by updating or adding the id of the sending node and the RSSI value of the received packet. Thus, the neighborhood discovery is independent of the control packet routing.

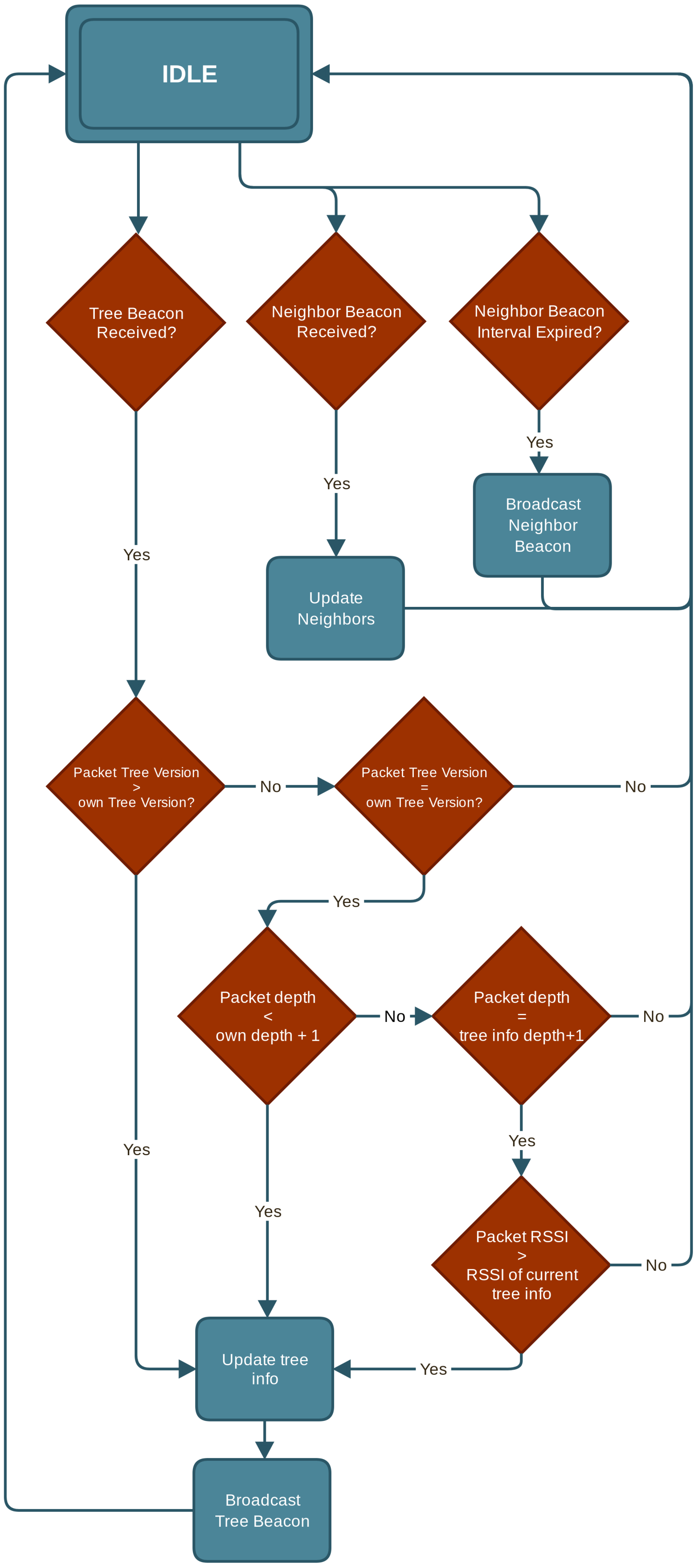

IDLE
Tree Beacon Received?
Neighbor Beacon Received?
Neighbor Beacon Interval Expired?
Yes
Broadcast Neighbor Beacon
Yes
Update Neighbors
Yes
Packet Tree Version > own Tree Version?
No
Packet Tree Version = own Tree Version?
No
Yes
Packet depth < own depth + 1
No
Packet depth = tree info depth+1
No
Yes
Yes
Yes
Packet RSSI > RSSI of current tree info
No
Update tree info
Yes
Broadcast Tree Beacon

FIGURE 3.11: Handle Beacon SDN-Wise (Updated)

Chapter 4

Distributed Routing Protocol Implementation

One aim of this thesis is to compare the performance of SDN-based routing algorithms with routing protocols that use distributed information for routing decisions. To perform this benchmark tests, we have implemented a version of Flooding and RPL that was deployable to the sensor nodes used in SDNWisebed. In this chapter the implementation of these routing protocols is sketched.

4.1 Flooding Protocol Implementation

A major part of this thesis was the comparison of common routing protocols with SDN-based routing protocols. It was the aim of this benchmark test to compare protocols in a scenario where nodes send packets to other nodes. This scenario is rather new in WSNs and, therefore, most protocols are optimized for other communication patterns. Flooding is a special case of a gossip protocol, which sends the information to all direct neighbors and without a delay before the transmission. Gossip protocols are very robust, as the information is spread over multiple paths, and, therefore, the nodes receive the information multiple times (Jelasity, 2013). It was designed to disseminate information to all nodes in a network and, thus, also sends the information to the designated destination node.

For our implementation of Flooding we used Contiki OS and built the algorithm on top of it. In our implementation a data message is broadcasted in a certain interval by the source node. Each node that receives this broadcast message is decreasing the TTL of the packet and rebroadcasts it. When the destination receives the broadcast message it prints the message. To reduce the overhead we have also added a message id to each packet. Each sensor node has a ring buffer that contains the source node id and the message id of packets it has broadcasted. Before every broadcast it is checked if this packet has already been broadcasted and if this is the case the broadcast is inhibited. With this inhibition it is ensured that every node does only broadcast each packet once and the overhead is significantly reduced.

Even if Flooding was not designed for node-to-node communication it is very reliable when only a few packets are transmitted. But the network is quickly overloaded when more packets are sent.

4.2 RPL Implementation

In the benchmark tests we compared the SDN-Wise (RSSI) and DTARP algorithms to RPL. RPL is the quasi standard for WSNs and is optimized for node-tosink communication patterns (Winter et al., 2012). To route the traffic from the source to the destination, RPL creates a tree structure. We have implemented a lightweight version of RPL to test our SDN-based algorithms against this protocol. The tree info consists of:

- tree version
- node depth
- parent node id
- parent rssi

Our implementation of RPL supports node-to-sink and node-to-node communication in storing mode. Therefore, each sensor node stores all of its subtrees. To build the tree structure of this RPL implementation, the sink node broadcasts a tree info packet. All sensor nodes that receive this tree version packet, check if they have already installed this tree version. If the tree version is newer than the tree version stored on the sensor node, then the tree version of the node is updated according to the information of the tree version packet. If the tree version is smaller than the tree version of the sensor node, then the packet is discarded. And if the tree version of the packet is equal to the tree version of the sensor node, then it is checked if the sensor node that broadcasted the tree version packet is a better candidate to be the parent of the processing sensor node. This is the case when by changing the parent node the path length to the sink node is decreased or when the path length to the sink node remains equal and the RSSI level of link to the candidate is better than the RSSI level of the link to the current parent. When a sensor node updates its tree version, it broadcasts its new tree version, so that the adjacent sensor nodes are informed about the update and can update as well when this new tree provides a better route to the sink node.

To update the tree version of a sensor node the flowing steps are executed:

- sensor node tree version = packet tree version
- node depth = packet node depth + 1
- parent node id = packet src id
- parent rssi = packet rssi

When this DODAG is built, each sensor node that has received the tree version, is able to send data packets to the sink node. To enable mobility of network nodes, the tree version is increased and issued in a certain interval by the sink node. The most recent tree version represents the most accurate available topology. The DODAG built by the tree versions enables node-to-sink communication but not node-to-node communication. To enable node-to-node communication more information is needed. RPL defines a storing and non-storing mode for downwards routing. In the non-storing mode, the sink node collects information about the complete tree and uses this information to send packets down the tree. Therefore, each packet is appended by a list of nodes that need to be passed until it reaches the target. This mode is convenient, when the sink node has less constraint hardware than the other nodes.

Especially when the sink node has more storage capacity, than the sensor nodes. This because no routing information needs to be stored on the sensor nodes. In our deployment the sink node and the sensor nodes are all TelosB motes and, therefore, the non-storing mode does not provide any benefit.

In our RPL implementation we used the storing mode. In this mode each sensor node stores all of its subtrees. Therefore, each node can route packets into its subtrees and when the destination of a packet is not part of a subtree, then the packet is sent to the parent node. This enables node-to-node routing. To store the subtrees on each sensor node we have crated an (nxn) matrix where each row represents the id of a sensor node and the node that can be reached over this sensor node. Consequently, this matrix consists of zero rows, if the sensor node does not have a neighbor. If the sensor node has a neighbor the row consists of the ids that belong to the subtree of the corresponding neighbor. To update this subtree information each sensor node is broadcasting a packet that contains all ids of the sensor nodes of its subtree. When this child id broadcast packet is received by a sensor node that does not have the sending node as parent, then all ids are added to the corresponding row in the subtree matrix. With this information it is possible to send packets from a source node to a destination node. To forward the packet the sensor node is checking if the subtree matrix contains the id of the destination node. If this is the case, the packet is sent to the id of the row where the destination id was found. If the destination id has not been found in the subtree matrix, the packet is sent to the parent node of the node that handles the packet.

To maintain the tree based routing of this RPL implementation two beacons are needed: the tree info and the subtree update. To ensure that the topology is accurate, these beacons need to be issued in a certain interval. This interval hardly influences the control overhead of the network and should be chosen according to the application needs. In this implementation we have used the same intervals we used for the SDN-based algorithms that have been compared with this protocol. Therefore, the results are comparable.

This RPL implementation was mainly designed for a benchmark test against SDN-based routing protocols. The protocol was designed to be used with TARWIS. Therefore, it suits well for a sensor network size of 40 sensor nodes but it is not scalable. When more sensor nodes are added, the (nxn) matrix would quickly grow over the storage capacity of the sensor node. This scalability issue can only be solved by a non storing mode RPL implementation, but this would also require to transmit each packet over the root node and would consequently result in less efficient packet forwarding.

For our implementation of the RPL protocol we used Contiki OS as operating system and implemented the protocol on top of it.

Chapter 5

Routing Protocol Design with SDNWisebed

A major advantage of SDN is the ability to create and test new networking applications in a rapid prototyping loop. This loop starts by identifying a problem that needs to be solved by a network application. When the problem is found the application can be implemented and tested in SDNWisebed. SDNWisebed provides a fast feedback to the researcher, so that he/she is able to adjust the application. This rapid prototyping should promote the development of new applications.

To evaluate SDNWisebed and the impact of SDN-based routing in WSN an experiment was created. In this experiment a new routing algorithm was designed to test if SDNWisebed enables rapid prototyping and to show the benefits of SDN-based routing. First, the challenges of WSN routing are highlighted and it is discussed how various routing protocols perform under this conditions. Later a dynamic traffic aware routing protocol is introduced to mitigate these challenges. The results of these experiments are then discussed in Chapter 6.

5.1 Challenge of WSN routing

Current Wireless Sensor Networks are ad-hoc networks that consist of a sink node and numerous sensor nodes. A sink node is a node connected to a border router to exchange packets with the Internet of Things and a sensor node is a node that can sense physical parameters and forward packets of other sensor nodes. In typical WSN deployments sensor nodes are sending sensor values towards the sink node and configuration packets are sent from the sink node to the sensor nodes. With this architecture the network traffic is sent over a tree like structure. The nodes closer to the root (sink node) must handle more traffic, because they have to handle the traffic between their subtrees. Consequently, the traffic is distributed unevenly over the network.

In recent years the sensor nodes become smarter and the requirements to sensor networks are changing. Smart sensor nodes might not only send sensor values to the sink node, but also need to exchange information with other nodes in the network. Consequently, sink-to-node and node-to-node communication become more important. With node-to-node communication and smart sensor nodes that might use adaptive sampling rates, the traffic patterns become unpredictable and dynamic. Due to the topology and possible different sampling rates, the traffic of node-to-node flows is uneven distributed over the nodes of the network.

The energy resources of sensor nodes are usually very limited, and most of the energy is needed for the radio (Antonopoulos et al., 2009). As the radio consumes most of the energy, a sensor node that handles more traffic will run out of battery earlier. Unfortunately, the most crucial nodes of the WSN tend to fail first, as they are either central or close to the root node and must handle most of the traffic.

Depending on the distribution of the traffic, the network might fail due to missing energy on crucial sensor nodes. Therefore, an even distribution of the traffic is key to maximize the lifetime of Wireless Sensor Networks. Additionally, a better distribution of traffic also reduces congestion and the packet loss related to congestion.

First, in this chapter the currently available routing protocols Flooding, RPL and SDN-Wise (RSSI) are analyzed with respect to their load distribution. Then, the goal for a new routing algorithm is formulated, its design is introduced and finally its implementation is discussed.

5.2 Analysis of the Traffic Distribution in Routing Protocols

In this section, the traffic distribution that results from the use of Flooding, RPL and SDN-Wise (RSSI) (see Section 2.8) is analyzed and discussed.

5.2.1 Flooding

As described in Section 2.8.1 the aim of gossip protocols is to send information to all nodes in a network (Jelasity, 2013). As Flooding disseminates the information to all nodes in the network, it is not the perfect match for a node-to-node communication. In the example discussed in Section 2.8.1 it can be seen that flooding has a huge overhead, even if each packet is only sent once at each node. With the Flooding protocol every node is involved in the transmission of each sent packet even if it is not the source, destination or an intermediate hop. Nevertheless, this protocol is easy to implement and will help us to understand the behavior of the other protocols, so it's worth to take a close look on it.

When Flooding is used, almost all nodes are participating in the delivery of all packets. If the packets are only sent once at each sensor node, then the traffic is almost perfectly distributed over the network. When all nodes have the same network activity the nodes will need the same amount of energy and will run out of battery approximately at the same time. However, the overhead is so huge, that the time when all the sensor nodes fail is the worst-case scenario. Consequently, a perfectly even distribution of the traffic is not always desired.

5.2.2 Routing Protocol for Low power and Lossy Networks (RPL)

As discussed in the beginning of this chapter the uneven distribution of traffic reduces the lifetime of the overall network. As explained in Section 2.8.2, RPL builds a DODAG to route packets from the sensor nodes to the sink node. When all nodes are sending the traffic to the same sink node, the nodes close to the sink have to handle more traffic and an unbalanced distribution of the network traffic is unavoidable. But when the sensor nodes have very uneven sampling rates or the branches of the

DODAG tree are of different size, then the distribution of traffic over the network gets even worse.

With the approach of RPL to send packets along a tree (see Section 2.8.2), the traffic for node-to-node communication is distributed very uneven over the network. If the non-storing mode of RPL is used, every packet is sent over the sink node. Hence, the sink node is involved in each transmission and also the nodes close to the sink node have to handle much more traffic than the nodes at the leaves of the routing tree. Due to the higher amount of traffic, a lot of congestion is to be expected close to the sink node. Additionally, with this approach the path length can be much longer than the shortest path when the depth of the tree is high.

Even if the storing mode of RPL seems to be promising for better distribution of the traffic and reduction of congestion close to the sink node, the following thought experiment shows that the effect is negligible: Assume we have a sensor network with n nodes and each node has a degree of k, then each node has k neighbors. Now let's assume that the sink node has k neighbors that are root nodes of subtrees with equal size. When we pick two nodes by random, then the probability is approximately 1/k that the nodes are in the same subtree. Consequently, in approximately 1-1/k of the cases the packet has to be sent over the root node, as the destination is not found within the same subtree. For example, if we assume that k=10, then 90% of the packets are still sent over the sink node and, therefore, the use of the storing mode can be questioned. Especially, the storing mode requires each resource constrained sensor node to store n*n routing entries. For a larger amount of sensor nodes, these n*n entries can quickly exceed the storage capacity of the sensor node.

5.2.3 SDN-Wise (RSSI)

The standard algorithm to forward packets with SDN-Wise uses shortest path routing with respect to the RSSI level of the links (see 2.8.3). Without the need of tree based routing to forward data packets the traffic is much better distributed over the network. But to provide all required information to the controller, the sensor nodes are sending report messages to the controller. The messages containing this information are all sent to the sink node using an RPL like tree-based routing. Therefore, a slightly uneven traffic distribution resulting out of the control messages can be observed. This tree based routing for the control packets is unavoidable, as the packets also need to find the controller when the SDN network was not built yet. The rate of the control messages can be adjusted and should be high for WSNs with a highly dynamic topology and low for WSNs where the topology rarely changes. Usually, the report rate of the sensor nodes can be set to a fraction of the sampling rate and, therefore, the unevenness of the traffic distribution resulting from the control packets can be neglected.

Even if the traffic is much better distributed over all the nodes in the Wireless Sensor Network, the SDN-Wise protocol using only RSSI values still has two major drawbacks: First, the topology of the WSN is not considered and, therefore, sensor nodes with a high node betweenness centrality (discussed later in Section 5.3.1) will face more traffic than others. Sensor nodes with a high node betweenness centrality have many shortest paths that are passing them, and consequently, more data flows with a shortest path routing are passing those sensor nodes. Second, the data rate of the different flows is not considered in the SDN-Wise (RSSI) routing. The traffic can be distributed unevenly over the network, as the data rate between some sensors can be much higher than between others.

Summarizing, the SDN-Wise (RSSI) protocol already improves the distribution of the traffic as it uses the shortest paths, but it is not specially designed for the task to distribute the traffic as even as possible to equalize the battery lifetime of the sensor nodes and thus, increase the overall network lifetime.

5.3 Dynamic Traffic Aware Routing Protocol for Wireless Sensor Networks (DTARP)

A major drawback of the protocols described in the previous sections is the unbalanced distribution of the network traffic or a very high overhead. The uneven distribution of the traffic is mainly the result of network hierarchies that are built on top of the ad-hoc network, the node centrality betweenness of the physically given network topology and the varying data rates of the different sensor nodes. To overcome, these issues we propose the DTARP.

In DTARP we use a SDN controller to route the traffic dynamically around central and heavily loaded nodes in order to improve the distribution of the network traffic and, therefore, increase the overall lifetime of the WSN. To route the traffic around these crucial points we designed a cost function that combines the node centrality betweenness (Equation 5.1) and dynamic traffic information in order to define the cost of a link. With this construction it is possible to increase the cost of routes through heavily loaded and central nodes and redirect the traffic over paths with unused capacities. Thus, the traffic is better distributed through the network.

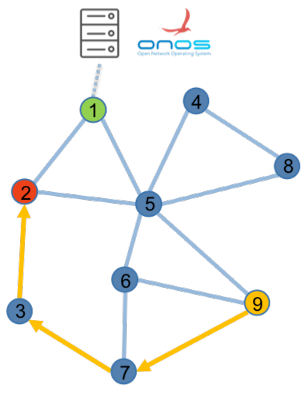


FIGURE 5.1: DTARP information dissemination

Figure 5.1 shows how a packet sent from one node to another could be routed, when it is more expensive to pass central or heavily loaded nodes. In this example network, most of the shortest paths between two nodes contain node 5. Hence, this node has the highest node betweenness centrality and it is very likely that it needs to handle more traffic than the other nodes. Consequently, the DTARP protocol tries to route the traffic around this node and the packet is sent along the border of the network.

5.3.1 Node Betweenness Centrality

The node betweenness centrality measures the centrality of a node in a graph (see eq. 5.1) (Barthélemy, 2004).

Let $G = (V, E)$ be the graph with the network nodes as vertices's V, the link between the nodes as edges E and n is the number of nodes. $\sigma(s,t|v)$ is the number of shortest paths between the nodes $s, t \in V$ which pass the node $v \in V$. $\sigma(s,t)$ is the total number of shortest paths between the nodes $s, t \in V$. Note, that it is possible to have multiple shortest paths of the same length between two nodes.

$$c_B(v) = \frac{1}{n(n-1)} \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)} \tag{5.1}$$

5.3.2 Traffic Cost

In our algorithm we define the traffic cost of a link as the average of the fraction of traffic handled by the two adjacent nodes. This is calculated by equation 5.2, where $\text{traffic}(v)$ denotes the number of packets received and sent by the node $v \in V$.

$$tf(e) = \frac{1}{2} \frac{\text{traffic}(v_1) + \text{traffic}(v_2)}{\sum_{u \in V} \text{traffic}(u)} \tag{5.2}$$

5.3.3 Cost Function

The cost function is defined by equation 5.3, where $e = (v_1, v_2)$, α is a constant to set the minimum cost of a link and β is a constant to adjust the weight of the dynamic part of the algorithm. In our experiment we used $\alpha = 0.01$ and $\beta = 0.5$. The value of the cost function is within the interval $cost(e) \in (\alpha, 1 + \alpha]$ as each known node has sent at least one report packet towards the controller and therefore $tf(e) > 0$. If the RSSI level of a link is below a certain threshold (in our experiments $threshold = 20$), the cost is set to the maximum to avoid a transmission over this link as the signal is not strong enough.

$$cost(e) = \begin{cases} 1 + \alpha & \text{if RSSI < treshold} \\ \alpha + \beta \cdot tf(e) + \frac{1}{2}(1 - \beta)(c_b(v_1) + c_b(v_2)) & \text{else} \end{cases} \tag{5.3}$$

Equation 5.3 provides a cost function which makes it more expensive to send packets over active and central nodes. To avoid oscillations, the static information of the node betweenness centrality was added, which provides a good estimate where most traffic is expected.

The SDN controller has a complete overview of the WSN topology and, thus, it is possible to use graph theory for routing decisions. To route packets with DTARP through the WSN, the Function 5.3 is used as a cost function for a Dijkstra shortest path algorithm evaluated in the SDN controller.

5.4 DTARP Implementation

In SDN-based networking the data plane is decoupled from the control plane. In a SDN managed Wireless Sensor Network the sensor nodes of the data plane are used for forwarding of network traffic and for sensing and actuating, and the controllers of the control plane are used for the routing decisions. To enable this decoupling of logic and forwarding, the sensor nodes need to exchange information with the controller. All this information handling is already implemented by the SDN-Wise framework. In the SDN-Wise framework each sensor node is sending reports with its local information to the controller. The controller uses this information for routing decisions. The routes are installed by a reactive routing approach. When a sensor handles a packet with a destination with no matching flow entry in its flow table, the sensor node is sending a request to the controller. The controller calculates the route and then returns a response or OpenPath packet. The response packet contains a flow entry with the next hop that must be taken to reach the destination. When the response packet is used to install the new path, each node on the path needs to request a flow entry on its own. The OpenPath packet is more efficient as it contains the complete path and is traversing the path and installing flow entries on each node of the path. Therefore, with the OpenPath packet the controller only needs to be consulted at the first node of a path (Dio et al., 2016; Galluccio et al., 2015a; Galluccio et al., 2015b).

In SDNWisebed an Open Networking Operating System (ONOS) server is used as controller. The SDN-Wise framework also contains an ONOS plugin that we used to implement DTARP. This ONOS plugin provides a Sensor Node Service, a Flow Rule Service and a Topology Service (Anadiotis et al., 2018; Anadiotis et al., 2015). Equipped with all those services, the researcher can focus on the implementation of the algorithm. The original SDN-Wise framework does not contain any dynamic traffic information in the report packets and, therefore, the controller has no traffic information for routing decisions. To provide the controller with this missing information, we have extended the report packets (see Chapter 3.2.3 for more details).

ONOS provides a very sophisticated plugin system based on the build system Maven. Maven is a build system with the aims to make the build process easy, providing an uniform build system, quality project information and guidelines for best practices development (*Maven – Introduction*).

To create a new ONOS networking application skeletons exists, and the developer can directly focus on the implementation of her/his work. All the services mentioned before can directly be accessed out of the ONOS application and, therefore, all information available to ONOS can be used in the application. To install the ONOS application on a running ONOS server it must be compiled to an *OAR* file, which can be installed in every running ONOS server. After the installation of the application, it can be activated and is ready to use.

To implement the DTARP algorithm we have created the ONOS application *onos-sdwsn-dynamic-routing*. This application implements the algorithm and shows the state of the sensor nodes in the web-interface. With all the services provided by the SDN-Wise framework combined with the updates we have documented in Chapter 3.2.3, the only thing missing for the dynamic routing is the routing algorithm itself. The controller has global information of the network, thus common graph algorithms can be used for the routing decisions. For example, the Dijkstra shortest path algorithm is very convenient to calculate the shortest path between two nodes. When each link is set to 1, the Dijkstra algorithm will always find the shortest path with respect to hops. But we can also use more sophisticated cost functions, like the DTARP we introduced above. The implementation of this kind of routing algorithm is quite easy. The Topology Service provided by ONOS manages and stores the complete WSN topology in a graph database. It implements several shortest path algorithms such as the Dijkstra algorithm and has knowledge of the topology. Only the cost function itself needs to be implemented and this is simply done by implementing the interfaces LinkWeigher shown in listing 5.2 and Weight shown in listing 5.1.

```
public interface Weight extends Comparable<Weight> {
    Weight merge(Weight otherWeight);
    Weight subtract(Weight otherWeight);
    boolean isViable();
    boolean isNegative();
}
```

LISTING 5.1: Weight interface that must be implemented

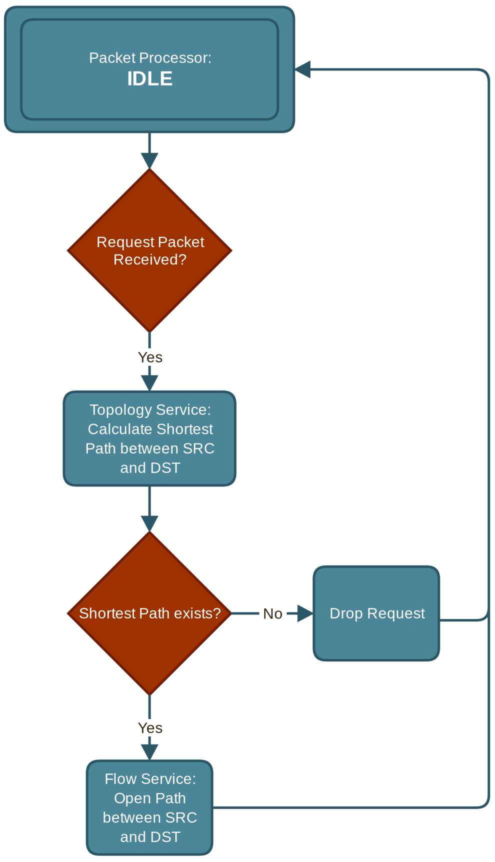
The Weight is used to represent the weight of links. In our case the cost of an edge is a scalar. Therefore, the implementation of the Weight interface is straight forward. The *merge* method should return the weight of the two weights, hence a new Weight with the sum of both Weights is returned. The *subtract* method is the same but with subtraction. *IsViable* should return true when a link can be used and false if not. In our algorithm each existing edge in the topology can be used and therefore, always true is returned. The *isNegative* method is always false as our cost function does not allow negative values. Additionally, the method *compareTo* of the extended interface Comparable must be implemented. This *compareTo* method just returns the value of the scalar comparison of both edges.

```
public interface LinkWeigher extends
   EdgeWeigher<TopologyVertex,TopologyEdge> {
    Weight weight(E edge);
    Weight getInitialWeight();
    Weight getNonViableWeight();
}
```

LISTING 5.2: LinkWeigher interface that must be implemented

The LinkWeigher is used to calculate the weight of a given edge. To avoid cycles in the routing a Weight object with value 0.1 is always returned in our implementation of the *getInitialWeight* method. The *getNonViableWeight* is null. The most important method for our algorithm is the *weight* method. It calculates the cost of a topology edge, and therefore, our previously introduced cost function is implemented here. The information of the Sensor Node Service can be used to get traffic statistics of sensor nodes adjacent to the edge. To calculate the node betweenness centrality we use the graphstream.org (*GraphStream - A Dynamic Graph Library*) library. With all these values available only trivial arithmetic is required to implement Equation 5.3. Consult Appendix A to see how LinkWeight and Weigher are implemented.

As shown before, the implementation is as easy as defining a weight with its arithmetic and a weigher to calculate the weight between two nodes. With the cost function implemented, only the handling of the request packets and the installation of


Packet Processor:
IDLE
Request Packet Received?
Yes
Topology Service: Calculate Shortest Path between SRC and DST
Shortest Path exists?
No
Drop Request
Yes
Flow Service: Open Path between SRC and DST

(A)

FIGURE 5.2: Handle Request Packet

the calculated flow entries is required. Figure 5.2 shows how an incoming packet is handled. To handle incoming packets we have implemented our own packet processor *DRPacketProcessor* and added it to the packet listener of the ONOS server. Thus, each incoming packet is also sent to the process method of *DRPacketProcessor*. In this packet processor it is checked if the packet is a request for a new flow entry. When it is a request the shortest path is calculated by the Topology Services with the use of the *DRLinkWeigher* and the *DRLinkWeight* class. The Topology Service returns a list of all nodes that need to be passed to get from the source to the destination of the request. With this path an OpenPath is sent back to the Sensor Node that sent the request. This sensor node will forward the OpenPath along the path to the destination and all nodes that process this packet will install the corresponding flow entries in their flow table. Consequently, only the first node needs to request an OpenPath packet to install a new route between two nodes.

This *onos-sdwsn-dynamic-routing* application's shortest path algorithm can also be used as a skeleton for other routing algorithms that are based on Dijkstra. For most algorithms that use a scalar cost function only the *weight* method of the *DR-LinkWeigher* must be changed. This change is very easy todo and in combination with the SDNWisebed testbed, it becomes easy to rapidly prototype various routing algorithms.

Chapter 6

Measurements and Evaluation

To evaluate SDNWisebed and DTARP various measurements have been done. This chapter covers the measurements and the evaluation of SDNWisebed and DTARP. First the topology of SDNWisebed is evaluated. Then a simple SDN application experiment with shortest path routing is discussed to show the benefits of SDN-based routing and the well functioning of SDNWisebed. The DTARP experiment shows the advantage of an application specific routing protocol and also the impact of SDN-based routing for Wireless Sensor Networks.

6.1 SDNWisebed: Topology

An important property of a WSN testbed is the topology of the sensor nodes. One option to change the topology is to change the transmission power of the sensor nodes. This section shows the measurements of theSDNWisebed topology when the transmission power is changed. In Figures 6.1, 6.2, 6.3, 6.4 the characteristics of the WSN topologies of different transmission power levels are shown.

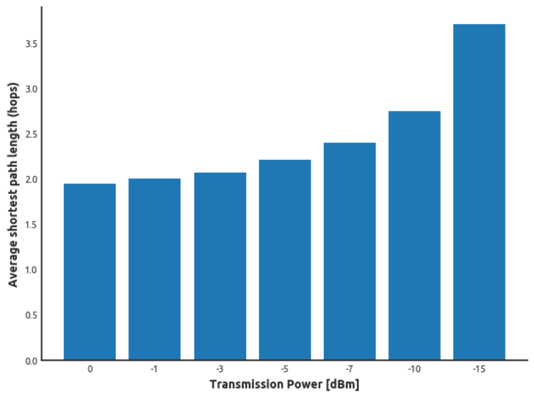


FIGURE 6.1: Average shortest path length

Figure 6.1 shows the average shortest path length in the network. The average shortest path length is defined in Equation 6.1 where V is the set of sensor nodes, $v_i, v_j \in V$, $n = |V|$ and $d(v_i, v_j)$ is the shortest path between the nodes v_i and v_j. This formula is valid for connected graphs.

$$l_S = \frac{1}{n(n-1)} \sum_{i \neq j} d(v_i, v_j) \tag{6.1}$$

In Figure 6.1, we can see that the length of the average shortest path is increasing when the transmission power is decreased. This result, was to expect as the signal range is decreasing when the transmission power is lowered. With the lower range of the signal more hops are needed to reach a target.

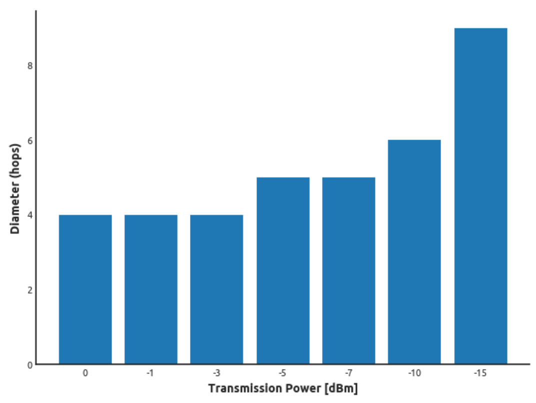


FIGURE 6.2: Diameter

Simultaneously, also the longest shortest path, the diameter, is increasing what is shown in Figure 6.2. The diameter is defined in equation 6.2.

$$\delta = \max_{i,j} d(v_i, v_j) \tag{6.2}$$

Figure 6.3 shows the average degree of a sensor node in the network. The average degree of the sensor nodes is defined in Equation 6.3, where $deg(v)$ is the degree of sensor node v. The reduced average degree is explained by the shorter signal range. When the transmission power is reduced and the signal range is decreasing, not so many nodes are reached by the signal anymore and consequently the average degree of the sensor nodes is decreasing.

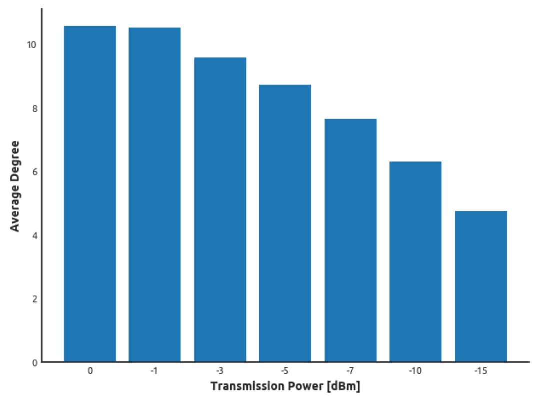


FIGURE 6.3: Average degree

$$deg_{avg} = \frac{1}{n} \sum_{v \in V} deg(v) \tag{6.3}$$

When the transmission power is decreased, the range of the signal is reduced and less neighbors are reached. Thus, we can see that the average degree is dropping, when the signal strength is decreased. The same is valid for Figure 6.4, where the maximum degree is shown in relation to the transmission power. The maximum degree is defined in Equation 6.4.

$$deg_{max} = \max_{v \in V} deg(v) \tag{6.4}$$

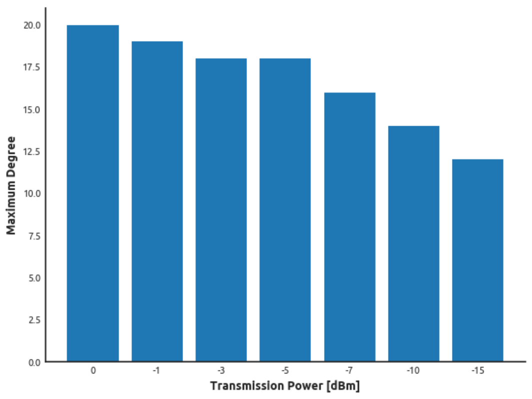


FIGURE 6.4: Maximum degree

For all transmission power levels shown in Figure 6.1 6.2 6.3 6.4, the network remained connected. Thus, there is a path between each pair of sensor nodes. It is also possible to reduce the transmission power even more, but then the network is not connected anymore and starts to cluster. Nevertheless, such a clustering of the network could be desired for some applications, which use multiple sinks in different clusters.

Table 6.1 shows the most important characteristics of SDNWisebed. In the network, we have some nodes which are very well connected and some which have only a few links. Overall the network is well meshed and all the nodes can be reached over a few hops. The longest shortest path (diameter) is 4, so each node can be reached within 4 hops from every node.

Nodes	40.
Max degree	20.
Min degree	2.
Average degree	10.6
Average shortest path	1.95
Diameter	4.

TABLE 6.1: TARWIS characteristics

In Figure 6.5 the distribution of the node betweenness centralities (see Equation 5.1) is shown. This figure shows, that there are some quite central nodes while others have very low betweennes centrality values. An unstructured deployment of the sensor nodes yields for a node betweenness centrality like this.

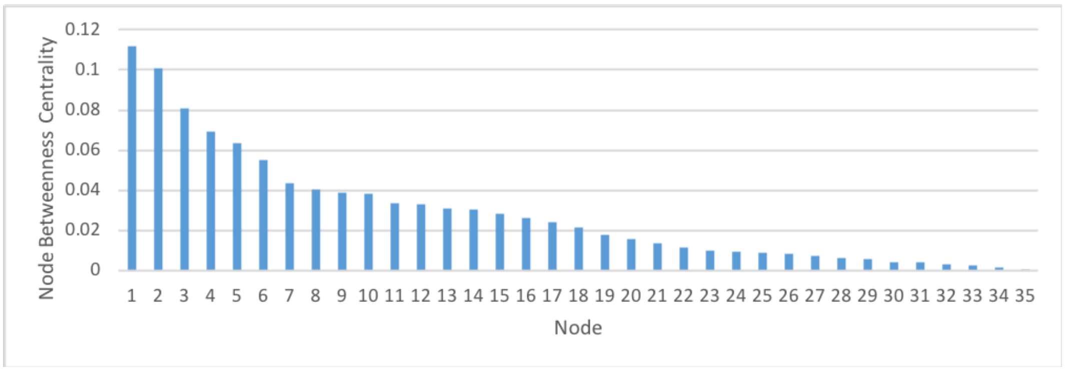


FIGURE 6.5: Distribution of node betweenness centrality

6.2 SDNWisebed Evaluation with Shortest Path Routing

To evaluate SDNWisebed we have created a simple test application using the SDN-Wise framework. In this application, one sensor node is sending a continuous stream of packets to another sensor node. We have created a simple routing application that calculates the shortest path according to the hop count. The packets are issued in 1s intervals. The period of the neighborhood discovery beacon and the interval of the report packets was set to 60s. We started the measurements 5 minutes after topology discovery was initiated and the complete topology of the WSN was discovered by the controller.

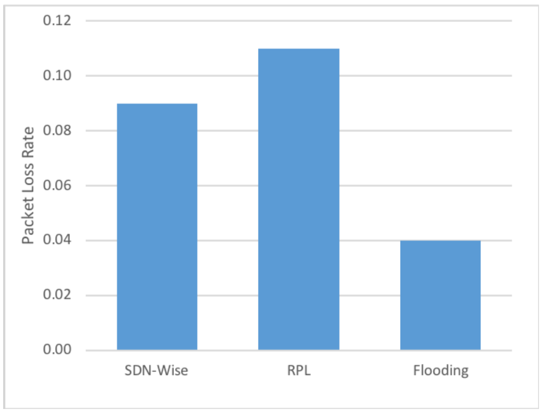


FIGURE 6.6: Packet Loss Rate

To evaluate the performance of this SDN-based routing protocol for sensor node to sensor node communication we have compared its performance with other common routing protocols. Therefore, we have implemented the Flooding protocol (Jelasity, 2013) and the Routing Protocol for Low-Power and Lossy Networks (RPL) with downward routes and non storing mode (Winter et al., 2012). As metric we used the packet loss rate as shown in Equation 6.7 and the number of packets sent in the network to maintain the network and forward the packet. The packet loss rate measures the fraction of sent data packets that never reached the destination. Figure 6.6 shows the packet loss rate we measured in this experiment. We measured a packet loss rate of 9% with the SDN-based shortest path routing, 11% with RPL and 4% with Flooding.

Packet loss is a common problem in WSNs and depends on the link quality, noise on the wireless channel and various environmental properties. Therefore, the packet loss is mainly indirectly linked to the routing protocols compared in this experiment. With each transmission there is a chance of a packet loss. Consequently, the likelihood of packet loss between source and destination is increasing with each hop the packet is sent over. As RPL does not use shortest path routing, it needs more hops to reach the target than the SDN-based shortest path protocol with SDN-Wise. Therefore, the packet loss rate is higher with RPL. Flooding only has a very low packet loss rate as the packet is sent over multiple paths and the packet reaches the destination multiple times.

$$PLR = 1 - \frac{\#\text{packets received}}{\#\text{packets sent}} \tag{6.5}$$

Figure 6.7 shows the number of packets sent each second with the different protocols. With a data packet rate of one packet each second, this is the traffic needed to forward a packet from its source to the destination. RPL and SDN-Wise both need reports and beacons for the maintenance of the routing. Even if those packets do not contain the same information, the overhead of these two packet types is equal, when the report and beacon period is set to the same length with RPL and SDN-Wise.

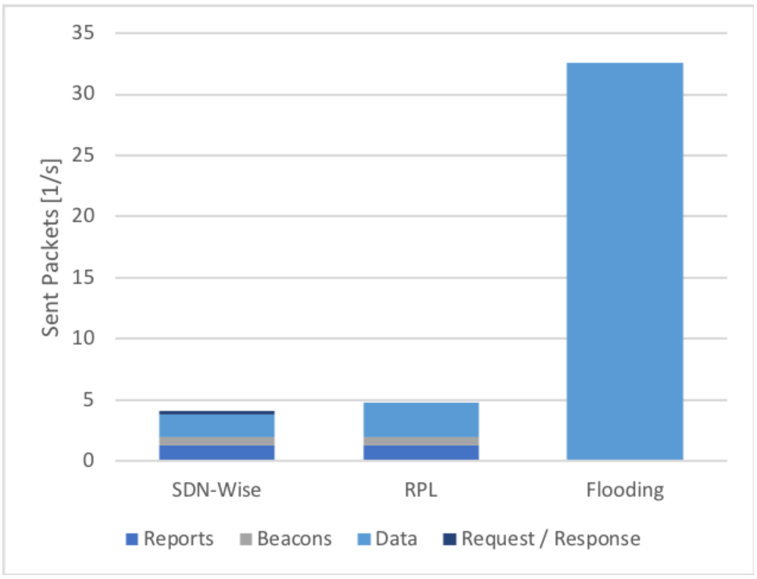


FIGURE 6.7: Packets sent per second in the whole network

By using SDN-Wise the flow routes needs to be installed in the network. This is done by a request packet, that is sent from a sink node, that misses a matching flow entry in its flow table, to the controller and the response packet the controller sends back to the requesting sensor node. This request only needs to be sent when no route has been established. The response or OpenPath packet replied by the controller to this request establishes a flow through the network and the following packets do not need to send a new request. Therefore, the request packets can be neglected if a larger amount of data packets is exchanged between two distinct nodes. The duration of this experiment was set to one minute and the flow was installed once during this time.

The significant difference in the number of sent packets can be found in the number of sent data packets. With SDN-Wise the shortest path between the source and the destination is found. In our experiment the shortest path between source and destination was 2 hops and, therefore, a data packet needs to be transmitted twice to reach its destination. RPL with downward routing does not use shortest path routing. In our experiment the data packet sent with RPL needed 3 hops to reach the destination and, consequently, more retransmissions were used than with the SDN-based shortest path routing. Using the Flooding protocol each of the sensor nodes that receives a packet is retransmitting the packet what results in a huge overhead in the data packet transmission with Flooding. Figure 6.7 shows the number of transmissions that are needed to transmit one packet from source node to destination node with this configuration.

6.3 DTARP

To prototype the DTARP algorithm we used two development stages. First, we used simulation to design the algorithm with a very fast feedback loop. When the algorithm was running as expected by using simulators we used SDNWisebed testbed to validate our results. The possibility fast deployment with SDNWisebed was a big help in the prototyping phase. With this fast feedback it was possible to apply iterative development strategies and use rapid prototyping. This section is split into two sections. First the simulation and its results are discussed and then the real-world measurements and results done with SDNWisebed are highlighted.

6.3.1 Simulation

To test and adjust DTARP the simulator COOJA was used. COOJA is a simulator for the Contiki OS sensor node operating system. COOJA allows for simultaneous simulation at the network level, the operating system level, and the machine code instruction set level (Osterlind et al., 2006).

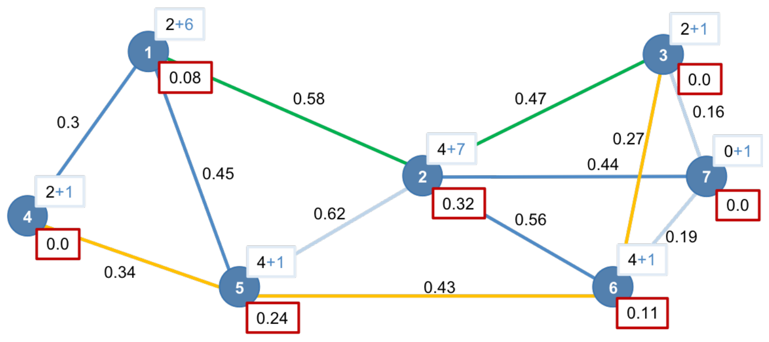
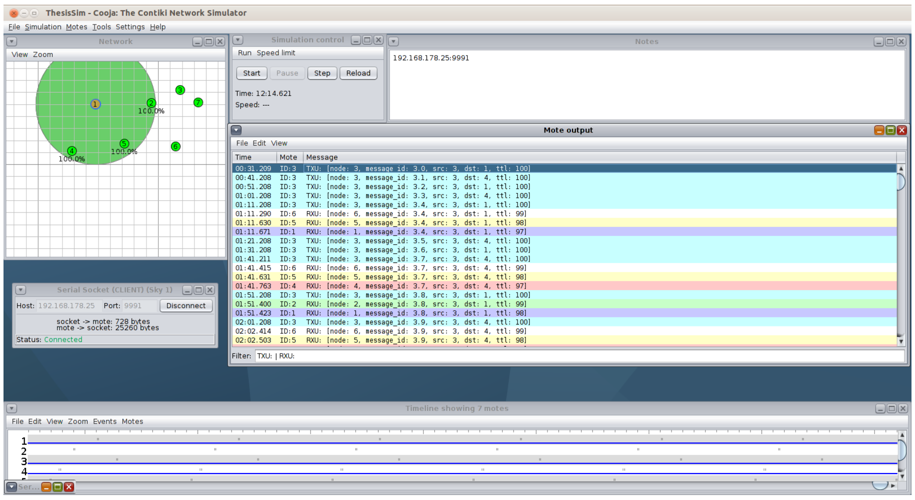


FIGURE 6.8: Simulation Topology

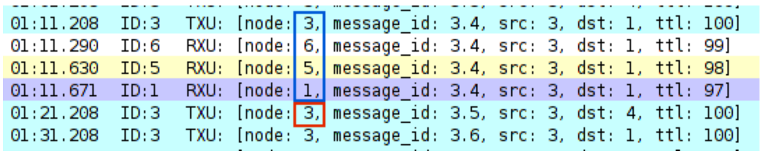
For this experiment we designed a network topology, that changes the routing during runtime when a certain number of packets was sent through the network. Figure 6.8 shows this topology and the expected routes when the routing has found the equilibrium of the path weights. In the red boxes the values of node betweenness centrality of the node is denoted and in the blue boxes the expected traffic through these nodes are shown. The traffic consists of the expected data traffic (black) and the expected control packets in blue per 10s. In this experiment node 3 sends data packets to node 1 and node 4 alternatively. The packets are issued in 10 seconds intervals.



```
Time       Mote   Message
00:31.209  ID:3   TXU: [node: 3, message id: 3.0, src: 3, dst: 1, ttl: 100]
00:41.208  ID:3   TXU: [node: 3, message_id: 3.1, src: 3, dst: 4, ttl: 100]
00:51.208  ID:3   TXU: [node: 3, message_id: 3.2, src: 3, dst: 1, ttl: 100]
01:01.208  ID:3   TXU: [node: 3, message_id: 3.3, src: 3, dst: 4, ttl: 100]
01:11.208  ID:3   TXU: [node: 3, message_id: 3.4, src: 3, dst: 1, ttl: 100]
01:11.290  ID:6   RXU: [node: 6, message_id: 3.4, src: 3, dst: 1, ttl: 99]
01:11.630  ID:5   RXU: [node: 5, message_id: 3.4, src: 3, dst: 1, ttl: 98]
01:11.671  ID:1   RXU: [node: 1, message_id: 3.4, src: 3, dst: 1, ttl: 97]
01:21.208  ID:3   TXU: [node: 3, message_id: 3.5, src: 3, dst: 4, ttl: 100]
01:31.208  ID:3   TXU: [node: 3, message_id: 3.6, src: 3, dst: 1, ttl: 100]
01:41.211  ID:3   TXU: [node: 3, message_id: 3.7, src: 3, dst: 4, ttl: 100]
01:41.415  ID:6   RXU: [node: 6, message_id: 3.7, src: 3, dst: 4, ttl: 99]
01:41.631  ID:5   RXU: [node: 5, message_id: 3.7, src: 3, dst: 4, ttl: 98]
01:41.763  ID:4   RXU: [node: 4, message_id: 3.7, src: 3, dst: 4, ttl: 97]
01:51.208  ID:3   TXU: [node: 3, message_id: 3.8, src: 3, dst: 1, ttl: 100]
01:51.400  ID:2   RXU: [node: 2, message_id: 3.8, src: 3, dst: 1, ttl: 99]
01:51.423  ID:1   RXU: [node: 1, message_id: 3.8, src: 3, dst: 1, ttl: 98]
02:01.208  ID:3   TXU: [node: 3, message_id: 3.9, src: 3, dst: 4, ttl: 100]
02:02.414  ID:6   RXU: [node: 6, message_id: 3.9, src: 3, dst: 4, ttl: 99]
02:02.503  ID:5   RXU: [node: 5, message_id: 3.9, src: 3, dst: 4, ttl: 98]
Filter: TXU: | RXU:
```

FIGURE 6.9: COOJA Overview

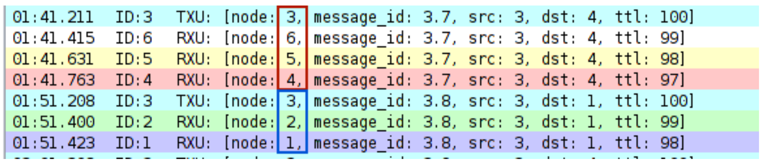
Figure 6.9 shows an overview of the COOJA simulator with the experiment. For the simulation we mainly used the topology overview of COOJA to place the simulated sensor node in order that they form the desired network topology. We use the log to evaluate the route of the packets. To evaluate if the algorithm finds the correct route each node that is forwarding a packet is printing a log entry. With this output it is possible to reproduce the path each packet has taken.



```
01:11.208  ID:3  TXU: [node: 3, message_id: 3.4, src: 3, dst: 1, ttl: 100]
01:11.290  ID:6  RXU: [node: 6, message_id: 3.4, src: 3, dst: 1, ttl: 99]
01:11.630  ID:5  RXU: [node: 5, message_id: 3.4, src: 3, dst: 1, ttl: 98]
01:11.671  ID:1  RXU: [node: 1, message_id: 3.4, src: 3, dst: 1, ttl: 97]
01:21.208  ID:3  TXU: [node: 3, message_id: 3.5, src: 3, dst: 4, ttl: 100]
01:31.208  ID:3  TXU: [node: 3, message_id: 3.6, src: 3, dst: 1, ttl: 100]
```

FIGURE 6.10: Initial route

Figure 6.10 shows the simulation at the beginning and how the initial packets were sent towards their destination. The packet with id 3.4 from node 3 to node 1 is sent over route (3,6,5,1), shown in the blue box. At this time not all sensor nodes have sent their reports yet and the controller has not discovered the complete topology. Therefore, for the packet with id 3.5, shown in the red box, no path to the destination was found and it was dropped.



```
01:41.211  ID:3  TXU: [node: 3, message_id: 3.7, src: 3, dst: 4, ttl: 100]
01:41.415  ID:6  RXU: [node: 6, message_id: 3.7, src: 3, dst: 4, ttl: 99]
01:41.631  ID:5  RXU: [node: 5, message_id: 3.7, src: 3, dst: 4, ttl: 98]
01:41.763  ID:4  RXU: [node: 4, message_id: 3.7, src: 3, dst: 4, ttl: 97]
01:51.208  ID:3  TXU: [node: 3, message_id: 3.8, src: 3, dst: 1, ttl: 100]
01:51.400  ID:2  RXU: [node: 2, message_id: 3.8, src: 3, dst: 1, ttl: 99]
01:51.423  ID:1  RXU: [node: 1, message_id: 3.8, src: 3, dst: 1, ttl: 98]
```

FIGURE 6.11: Route in equilibrium

Figure 6.11 shows the flow of the packets when the dynamic routing protocol has found its equilibrium and the packets are sent over stable paths. The packet with id 3.7 from node 3 to node 4 is sent over route (3,6,5,3) shown in the red box and packet with id 3.8 from node 3 to node 1 is sent over route (3,2,1) shown in the blue box.

Here the paths are the same as we expected in Figure 6.8, when we calculated the equilibrium for DTARP. This shows that the algorithm is behaving as expected and that the algorithm is ready for real-world tests with SDNWisebed.

6.3.2 DTARP Measurements with SDNWisebed

This Section explains how we used the SDNWisebed for real world evaluation of the routing algorithm DTARP and the measurements are shown.

Scenarios

In Chapter 5 we motivated why node to node communication is crucial for the Internet of Things. Therefore, we designed two scenarios to measure the performance of the different protocols for point to point communication.

In the first scenario, one node is sending a continuous stream of packets towards another node. The source and destination nodes are selected randomly, but the same nodes were used for all measurements. The shortest path (by hops) between the source and the destination node was two hops in the first scenario.

In the second scenario, a random permutation was chosen to assign a destination node to each node. Each node sends a continuous stream of packets towards the node which was assigned as its destination. Thus, we have 40 active connections in this experiment.

Metrics

The examined protocols differ in how they route the data packets to the destination. So we expect the packets to take different paths with each of the protocols. These paths vary in their length. Therefore, we used the average hop count (see equation 6.6) to compare the protocols. In equation 6.6 P is the set of sent packets, $hops\_at\_dst(p)$ is the number of hops the packet p has taken from the source to the destination and $\#P$ is the number of sent packets.

$$\text{Average hop count} = \frac{\sum_{p \in P} hops\_at\_dst(p)}{\#P} \tag{6.6}$$

Another important metric is the packet loss rate (see equation 6.7). It is used to measure the fraction of packets which were sent but did not reach the destination. In the measurement only the data packets were considered in the packet loss rate.

$$\text{Packet loss rate} = 1 - \frac{\text{\# received data packets}}{\text{\# sent data packets}} \tag{6.7}$$

As motivated in the beginning of this chapter a good distribution of the traffic is key to maximize the lifetime of a sensor network. To compare the different protocols the traffic distribution function $tf(v)$, where the nodes v are ordered according to their load, can be used (see equation 6.8).

$$tf(v) = \frac{traffic(v)}{\sum_{v \in V} traffic(u)} \tag{6.8}$$

Results

In this section the results of the benchmark tests are shown. These results show a clear difference between the tested protocols. All protocols behaved as expected when only one node was sending packets (1 connection) towards one destination. The shortest path between the sending and receiving node was 2 hops. Figure 6.12 shows that Flooding and the SDN-Wise with (RSSI) routing sent the packet over the shortest path while RPL and DTARP sent the data packets over paths longer than the shortest path. In RPL the packets are routed over the tree shaped overlay network towards the root node, until the destination node is contained in a subtree of an intermediate node and then routes them towards the destination. Thus, not the shortest path is taken. DTARP avoids to send packets over central nodes and nodes that handle a lot of traffic. Therefore, the path to send the packets is longer than the shortest path.

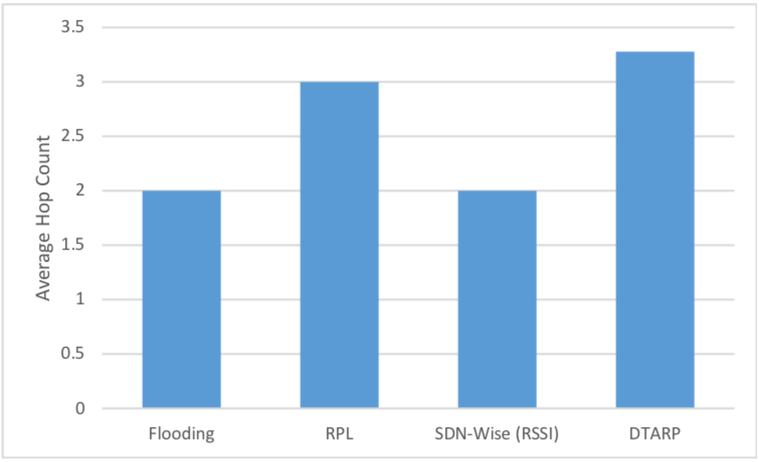


FIGURE 6.12: 1 Connection Average hop count

Figure 6.13 shows that RPL, SDN-Wise (RSSI) and DTARP have almost the same packet loss rate. With the Flooding protocol packets are sent over multiple paths towards the destination, so the packets reach the destination multiple times. Therefore, Flooding has a lower packet loss rate than the other protocols.

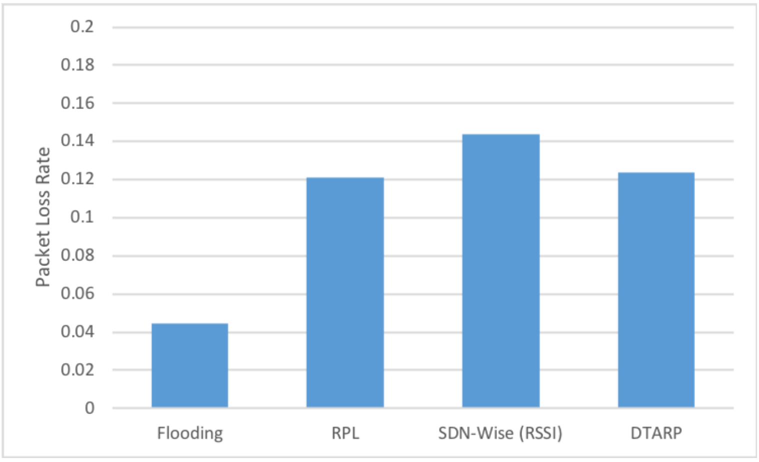


FIGURE 6.13: 1 Connection Packet loss rate

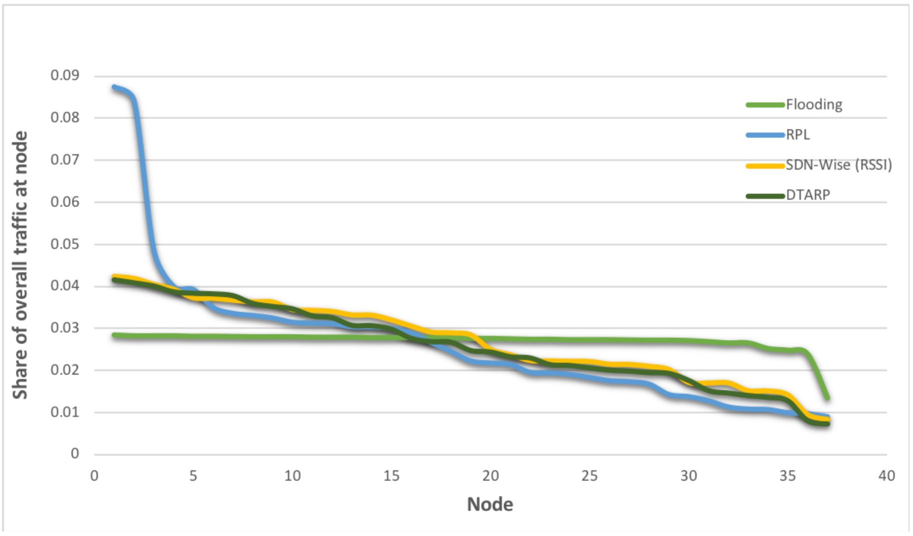


FIGURE 6.14: Traffic distribution with 1 connection

Figure 6.14 shows the share of all traffic handled by a node. The share of the traffic handled by a node is the ratio of all packets that were sent in the network to the packets handled by the node. Note that the nodes are ordered by the node activity. With Flooding each node retransmits the data packets. Therefore, almost all nodes participate in each transmission and share the same activity. This result is reflected by Figure 6.14 where only a few nodes on the border of the network do not retransmit all packets and therefore, have a lower network activity. If RPL is used, a few nodes are much more active than others; this was expected as all the control packets and most of the data packets are sent over the root node. Due to the control packets to maintain the SDN network, which are exchanged between the controller and the sink nodes to maintain the network, the SDN-Wise based protocols DTARP and SDN-Wise (RSSI) show a slightly unbalanced linear distribution of the traffic. In this experiment, where only one sensor node is sending data packets to one other sensor node, the fraction of the data packets is so small that the control packets predominate. As DTARP and SDN-Wise mainly differ in data packet routing, both protocol have a similar distribution.

In the second scenario 40 random node pairs are built. The pairs are built with a random but fixed permutation of the nodes. For each pair, one node is the source and the other the destination. The source nodes send a continuous stream of packets towards their allocated destination. Therefore, 40 connections are built in the network. In this scenario much more data packets are exchanged,the network is much more congested than it the previous experiment, and the network is on its limit. Flooding did not work in this scenario, as the sensor nodes where not able to handle all traffic, which was the result of the retransmissions. Therefore, Flooding is not shown in the results of this scenario. In Figure 6.15, the hop count of the sent messages from the source to the destination with the RPL protocol is higher than the average shortest path length discussed in section 6.1. This is the consequence of sending all messages along the tree shaped routing of RPL. Having so many packets sent over a tree shaped overlay network, results in congestion on the nodes close to the root node. When congestion occurs on a node, it is not able to handle all traffic and packets are dropped. Therefore, the packet loss rate of RPL in Figure 6.16 is quite high.

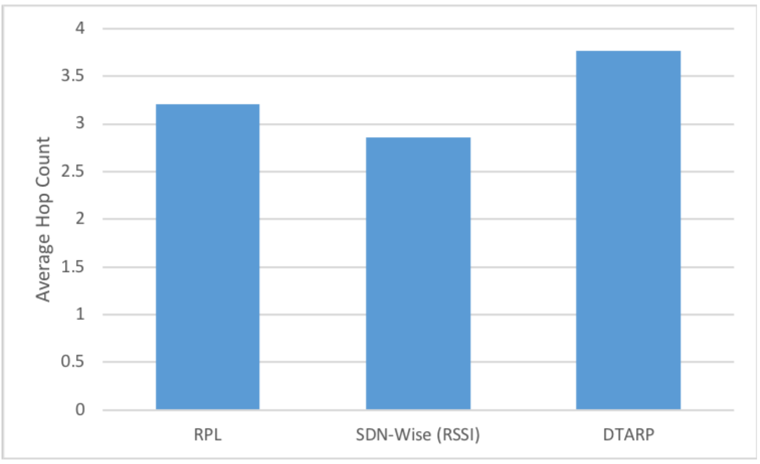


FIGURE 6.15: 40 Connections average hop count

In Figure 6.15, SDN-Wise (RSSI) has slightly higher average hop count than the average shortest path of SDNWisebed (see Section 6.1). This is because the Radio Signal Strength Indicator values of the received beacon packets are used in the shortest path optimization, what does not always lead to the shortest path in terms of hops.

As the messages are routed around active and central nodes, the DTARP routing protocol results in a higher average hop count than the average shortest path of the network.

In the second experiment 40 sensor nodes are sending packets to one distinct other sensor node. Thus, in the second experiment much more traffic was generated than in the first where only 1 sensor node was sending packets to one other sensor node. Therefore, the congestion in the network was higher in the second experiment. With higher congestion the packet loss rate also increases. Figure 6.16 shows the packet loss rates of the multi connection scenario. The packet loss rate for the SDN-Wise based protocols is much lower than the packet loss rate of RPL. This is because the traffic is much better distributed and not all traffic is routed over the most congested node.

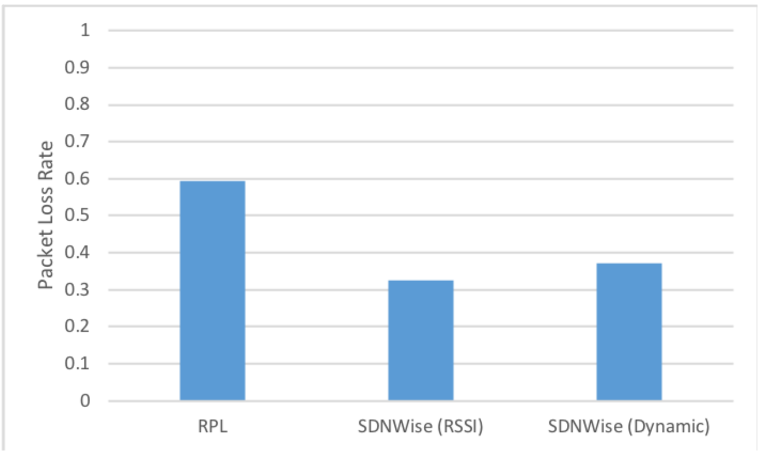


FIGURE 6.16: 40 Connections packet loss rate

The tree based structure of the RPL-like protocol can clearly be seen in Figure 6.17. With this protocol almost a quarter of all packets are sent over the root node. Therefore, the leaf nodes of the tree only contribute to a small fraction to the overall traffic.

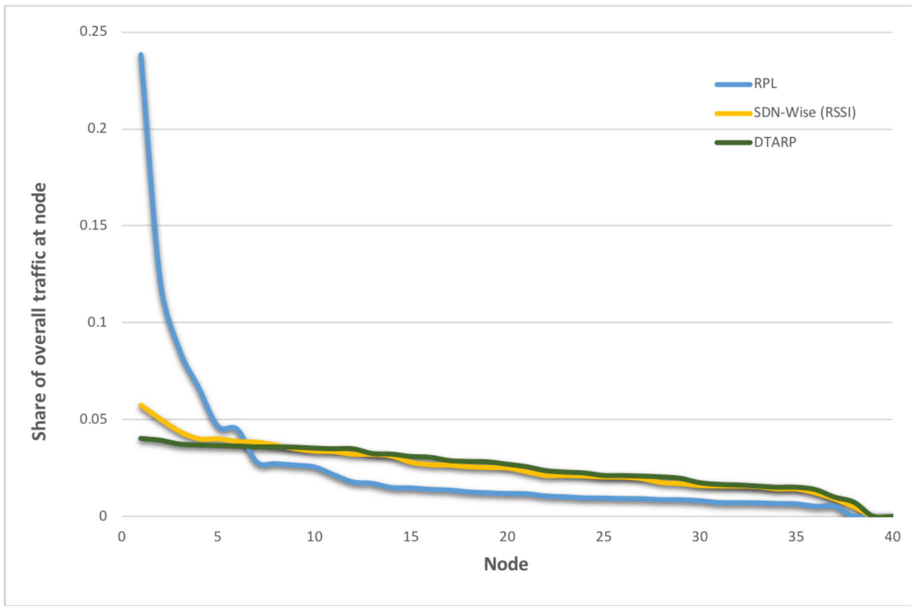


FIGURE 6.17: Traffic distribution with 40 connections

In Figure 6.17 the SDN-Wise (RSSI) protocol shows a much better distribution of the traffic. But it can still be recognized, that the most active two nodes handle up to a quarter more traffic than the next most active nodes. These nodes are the most central nodes where most of the shortest paths are passing by and, therefore, also the most traffic is routed through.

By the use of the DTARP routing, the most active nodes could be relieved by routing the traffic around them. For the other nodes a load distribution like in SDN-Wise (RSSI) can be observed.

Chapter 7

Discussion

In this thesis SDNWisbed a Software-Defined Wireless Sensor Network Testbed was designed and implemented. This testbed allows rapid prototyping for SDN applications and should promote the development of new SDN applications. To show the capabilities of SDNWisebed and the advantages of SDN-based routing in WSNs DTARP a Dynamic Traffic Aware Routing Protocol was designed, implemented, and tested in this thesis.

In the literature study in Chapter 2 we found three testbeds for SDN-Wise. These testbeds were mainly designed for specific applications and, therefore, not suitable for rapid prototyping.

Compared to these testbeds, SDNWisebed introduced in Chapter 3 has some significant benefits. The EuWIn testbed and the testbed used by Million B were mainly designed to test one specific application and therefore, they do not promote rapid prototyping. SDNWisebed provides rich types of network management functions, such as resource reservation features, support for reprogramming and reconfiguration of the nodes, provisioning to debug and remotely reset sensor nodes in case of node failures, as well as a solution for collecting and storing experimental data. Therefore, SDNWisebed can be used for rapid prototyping of SDN-based networking applications.

To build SDNWisebed we have extended the SDN-Wise framework by several functions and improvements. Those changes are documented in Chapter 3. By adding dynamic information to the report packets of SDN-Wise we have enabled dynamic routing. The addition of the ability of dynamic routing is an important step towards traffic engineering in Wireless Sensor Networks. The improvement of the control packet routing makes SDN-Wise more stable in real world deployments.

In Chapter 5 we explained that the distribution of the traffic in the common Wireless Sensor Network routing protocols is very uneven and some sensor nodes have to handle much more traffic than others. These nodes also need much more energy and consequently, run out of battery earlier. For example, in a RPL node-to-node communication every packet needs to be sent along the routing tree. In Chapter 5 we could explain that with this approach almost every packet needs to be sent over the root node. This leads to a very unbalanced distribution of the traffic and is therefore, very inefficient in terms of energy consumption. In Chapter 5 we introduced DTARP, a routing algorithm that solves this problem by better distributing the traffic over the nodes. With the SDNWisebed testbed we could test the algorithm in a real-world environment. The benchmark test of the Flooding, RPL, SDN-Wise (RSSI) and DTARP algorithm clearly showed the advantage of SDN-based algorithms for node-to-node communication. Flooding is overloading the network by definition and therefore, should only be used to disseminate information to all nodes information but not for node-to-node communication.

In Chapter 6 the results of this thesis are shown. First the topology of SDNWisebed is evaluated. The evaluation of the topology shows that SDNWisebed allows to build WSN topologies by changing the transmission power of the sensor nodes. Therefore, it is possible to test network applications in various but repeatable environments. Next in Chapter 6 the performance of SDN-based applications is tested in SDNWisebed. In a first experiment a SDN-based shortest path by hop count algorithm was compared to RPL and Flooding. This experiment showed that the SDN-based algorithm is better than RPL and Flooding according to overhead. The shortest path algorithm also outperforms RPL according to packet loss rate. In this experiment Flooding has the lowest packet loss rate as the packets are sent to all sensor nodes and, therefore, the packets reach their destination multiple times. However, the overhead generated by Flooding is so huge that Flooding is not applicable in networks with a higher data rate. DTARP is evaluated and compared against Flooding, RPL and SDN-Wise based shortest path routing. This evaluation showed that RPL has a very unbalanced distribution of the node activity and some nodes handle much more traffic than others. With SDN-Wise (RSSI) and DTARP the distribution of the sensor node radio activity is distributed much better. This chapter also showed that by the use of DTARP the traffic load of the two most active sensor nodes was reduced by up to 25% compared to the SDN-Wise (RSSI). So, we could show that SDN significantly boosts the routing in WSNs when sink to node or node to node communication is needed. For node to sink communication RPL is very powerful and therefore, it is difficult to compete with RPL in this mode.

7.1 Conclusions

Even if some prototypes of SDN frameworks for WSNs exists, SDN-based WSN applications are still rare. A major reason for the reluctant progress of SDN-based WSN applications is the lack of application and research project independent testbeds that enable rapid prototyping of new applications. To overcome this problem we have built SDNWisebed a Software Defined Networking Testbed for Wireless Sensor Networks. To build a testbed for SDN-based WSN applications several components are needed. Most important are a SDN framework, with controller and sensor node firmware and a management system to deploy and schedule experiments. This thesis showed how to extend the WSN management system TARWIS by the SDN framework SDN-Wise to build the SDN enabled WSN testbed SDNWisebed.

In this thesis we could show that the design of a new SDN-based routing protocol is very efficient when SDNWisebed is used. To build a new SDN-based routing protocol, first a routing problem needs to be identified and then it is easily solved by the implementation of a cost function for a Dijkstra algorithm running on the SDN controller. The evaluation of this protocol is done by the analysis of generated log files.

The main benefit that the SDN-based routing provides to WSNs, is the global topology view of the SDN controller. This global view of the WSN topology enables various opportunities for new routing protocols. Because WSNs are meshed networks and, therefore, lack a physical hierarchy, this global view is very valuable for node-to-node communication. However, a lot of WSN applications only need node-to-sink communication where RPL is very efficient and outperforms the SDN-based WSN routing protocols. But with the sensor nodes becoming smarter and smarter the node to node communication pattern also gains importance in WSNs.

The SDN-based routing protocol DTARP clearly showed that there is an advantage of SDN-based routing protocols over a distributed routing protocol with respective to energy efficiency. With the global topology view of the controller it is possible to route the traffic in a manner that not a few nodes have to handle most of the traffic and, therefore, run out of battery first. With the better distribution of the traffic the overall network lifetime is increased and therefore, the energy efficiency of the WSN improved.

7.2 Future Work

In the following some topics are highlighted that might be an interesting starting point for further research.

Within this work a SDNWisebed a Software-Defined Wireless Sensor Network Testbed has been created and evaluated. The evaluation of this testbed showed, that SDN can significantly boost the performance of WSN applications and that this testbed enables rapid prototyping for WSN applications. The literature study also showed that not many SDN-based applications for WSNs are available yet. Consequently, the future work will be to design SDN-based applications for WSNs and use SDNWisebed to test and evaluate these applications.

The vision of Internet of Things requires that different devices of numerous manufacturers are able to exchange information with each other. As Wireless Sensor Networks are an important component of Internet of Things, they should seamlessly integrate into the IoT. Currently, SDNWisebed is still more an Intranet of Things than part of the Internet of Things as it is not connected to any other WSN or IoT infrastructure. SDN for WSN is a promising approach to overcome the heterogeneity of the devices of different manufacturers. To test how much SDN can leverage the interoperability of different devices it should be better connected to the Internet of Things. For example a framework like VICINITY (Guan et al., 2017) can be used to connect the sensor nodes of SDNWisebed to other IoT infrastructure.

A WSN can consist of several different sensor nodes. Sensor nodes that become smarter and smarter are not restricted to sensing anymore. Today a WSN can consist of sensor nodes like temperature sensor and actuators like thermostats. Therefore, the nodes in a WSN should be able to exchange information. A central registry on the SDN controller could be developed that allows sensors to register the properties they produce and actuators can subscribe to properties they consume. The thermostat should then be able to request all temperature values measured in the room it is managing the temperature.

With the fast growing number of IoT devices connected to the Internet the security of all devices that are connected to the Internet becomes more and more important. The security of SDN in WSNs has only been rarely researched and the current implementation of the SDN-Wise framework does not contain any security layer. To become a better understanding of the security in SDN managed WSNs more research is urgently needed. Therefore, potential attack vectors need to be identified and protection mechanisms against these attack vectors are required. This research should focus on the very different properties of the three main components of the SDN managed WSN, the controller, the border router and the sensor nodes. A failing controller or border router might be very harmful for the network but a failing sensor node might not matter.

Within this thesis we have extended SDN-Wise by dynamic information to enable dynamic routing and traffic engineering. The implementation of the update of existing flows is currently quite static. By improving this update procedure also the control packet overhead could be reduced. The update could be improved by a trickle timer to decide when to drop a packet or by artificial intelligence that decides when a flow is not used anymore. But as all these approaches are based on statistical estimations those methods need to be benchmarked against each other. The current method of dropping the flows in a certain interval already tested quite well and it might be a challenge to compete against.

Mobility is an important property of Wireless Sensor Networks. But with mobility also the WSN topology changes continuously. In a SDN managed network topology changes require continuous update of the flows in a network. When updates are not executed fast enough packets are lost. The controller could be used to predict the mobility of the WSN and install the routes in a way that it is able to minimize the packet loss.

Appendix A

Implementation of DTARP Cost Function

A.1 Implementation of the Weight Interface

```
public class DRLinkWeight implements Weight {

    private double cost;

    public DRLinkWeight() {
        this.cost = 1.0; // set the initial weight to 1
    }

    /**
     * Create a new linkweight by a skalar weight
     * @param cost
     */
    public DRLinkWeight(double cost) {
        this.cost = cost;
    }

    /**
     *
     * @param weight
     * @return the sum of this weight and the merged weight
     */
    @Override
    public Weight merge(Weight weight) {
        Weight tmpWheight = new DRLinkWeight(
                ((DRLinkWeight)weight).getCost() +
                    this.cost);
        return tmpWheight;
    }

    /**
     *
     * @param weight
     * @return the difference of this weight and the
        subtracted weight
     */
    @Override
    public Weight subtract(Weight weight) {
        Weight tmpWheight = new DRLinkWeight(
                this.cost -
                   ((DRLinkWeight)weight).getCost());
        return tmpWheight;
    }

    /**
     * @return true -> all established links are viable
     */
    @Override
    public boolean isViable() {
        return true;
    }

    /**
     *
     * @return false when cost is smaller than 0
     */
    @Override
    public boolean isNegative() {
        return cost < 0;
    }

    /**
     *
     * @param o the weight to compare to
     * @return -1 when this is smaller than o
     *         0 when this is eqal to o
     *         1 when this is greater than o
     */
    @Override
    public int compareTo(Weight o) {
        return (new Double(cost)).compareTo(
                new Double(((DRLinkWeight)o).getCost()));
    }

    /**
     *
     * @return cost as scalar
     */
    public double getCost() {
        return this.cost;
    }

    /**
     *
     * @param cost as scalar
     */
    public void setCost(double cost) {
        this.cost = cost;
    }
}
```

A.2 Implementation of the LinkWeigher Interface

```
public class DRLinkWeigher implements LinkWeigher {

    private final Logger log = getLogger(getClass());

    private static final double ALPHA = 0.1;
    private static final double BETA = 0.5;
    private static final double RSSI_THRESHOLD = 20;

    SensorNodeService sensorNodeService;
    SensorNodeStore sensorNodeStore;
    TopologyService topologyService;

    public DRLinkWeigher(SensorNodeService
       sensorNodeService, SensorNodeStore sensorNodeStore,
                         TopologyService topologyService) {
        this.sensorNodeService = sensorNodeService;
        this.sensorNodeStore = sensorNodeStore;
        this.topologyService = topologyService;
    }

    /**
     * Calculates the weight of the topology edge
     * @param topologyEdge
     * @return weight of the topology edge
     */
    @Override
    public Weight weight(TopologyEdge topologyEdge) {

        DeviceId srcNodeId =
           topologyEdge.link().src().deviceId();
        DeviceId dstNodeId =
           topologyEdge.link().dst().deviceId();

        topologyEdge.link().annotations().keys().stream()
                .forEach(key -> log.info(key));


        SensorNode srcNode =
           sensorNodeService.getSensorNode(srcNodeId);
        SensorNode dstNode =
           sensorNodeService.getSensorNode(dstNodeId);
        int RSSI =
           sensorNodeStore.getSensorNodeNeighbors(srcNode.id())
```

```
                .get(dstNode.id()).getRssi();

        double trafficOnLink =
           (sensorNodeStore.getSensorNodeNeighbors(srcNode.id())
                .get(dstNode.id()).getRxCount()
            +
               sensorNodeStore.getSensorNodeNeighbors(srcNode.id())
                .get(dstNode.id()).getTxCount()
            +
               sensorNodeStore.getSensorNodeNeighbors(dstNode.id())
                 .get(srcNode.id()).getRxCount()
            +
               sensorNodeStore.getSensorNodeNeighbors(dstNode.id())
                 .get(srcNode.id()).getTxCount()
            )/2.0;

        int overallTraffic = overallTraffic();

        double edgeBetweennessCentrality =
           getEdgeCentralityBetweenness(srcNode, dstNode);

        double cost = ALPHA +
           BETA*trafficOnLink/overallTraffic + (1-BETA) *
           edgeBetweennessCentrality;

        return new DRLinkWeight(cost);
    }

    /**
     * Calculates the edge centralityBetweenness as
        avarage of the two corresponding nodes
     * @param src node
     * @param dst node
     * @return edgeCentralityBetweenness
     */
    public double getEdgeCentralityBetweenness(SensorNode
       src, SensorNode dst) {

        TopologyGraph topologyGraph = this.topologyService
                .getGraph(this.topologyService.currentTopology());
        Graph graph = new SingleGraph("TopologyGraph");
        graph.setAutoCreate(true);
        graph.setStrict(false);

        for(TopologyVertex node :
           topologyGraph.getVertexes()) {
            graph.addNode(node.deviceId().toString());
        }

        for(TopologyEdge link : topologyGraph.getEdges()) {
            String edgenName =
               link.src().deviceId().toString() + " - " +
               link.dst().deviceId().toString();
            graph.addEdge(edgenName,
                   link.src().deviceId().toString(),
                   link.dst().deviceId().toString(),
                      false);
        }

        BetweennessCentrality bcb = new
           BetweennessCentrality();
        bcb.setUnweighted();
        bcb.init(graph);
        bcb.compute();

        double centrality = 0;
        for(Node node : graph.getEachNode()) {
            centrality += (double)node.getAttribute("Cb");
        }

        Node a = graph.getNode(src.deviceId().toString());
        Node b = graph.getNode(dst.deviceId().toString());
        return ((double)a.getAttribute("Cb") +
           (double)b.getAttribute("Cb"))/(2*centrality);
    }

    /**
     * Returns the number of sent packets in the last
        period
     * @return overallTraffic
     */
    public int overallTraffic() {
        int traffic = 0;
        for(SensorNode srcNode :
           sensorNodeStore.getSensorNodes()) {
            for(SensorNodeNeighbor neighbor :
               sensorNodeStore
                    .getSensorNodeNeighbors(srcNode.id()).values()){
                traffic += neighbor.getRxCount() +
                   neighbor.getTxCount();
            }
        }
        return traffic;
    }

    /**
     *
     * @return initial weight
     */
    @Override
    public Weight getInitialWeight() {
        return new DRLinkWeight(this.ALPHA);
    }

    @Override
    public Weight getNonViableWeight() {
        return null;
    }
}
```

Bibliography

Abrignani, M. D. et al. (2013). “The EuWIn Testbed for 802.15.4/Zigbee Networks: From the Simulation to the Real World”. In: *ISWCS 2013; The Tenth International Symposium on Wireless Communication Systems*, pp. 1–5.

Anadiotis, A. C. G. et al. (2015). “Towards a software-defined Network Operating System for the IoT”. In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pp. 579–584. DOI: 10.1109/WF-IoT.2015.7389118.

Anadiotis, Angelos-Christos G. et al. (2018). “Towards Unified Control of Networks of Switches and Sensors through a Network Operating System”. In: *IEEE Internet of Things Journal*, pp. 1–1. ISSN: 2327-4662. DOI: 10.1109/JIOT.2018.2805191. URL: http://ieeexplore.ieee.org/document/8289363/.

Antonopoulos, C. et al. (2009). “Experimental evaluation of a WSN platform power consumption”. In: *2009 IEEE International Symposium on Parallel Distributed Processing*, pp. 1–8. DOI: 10.1109/IPDPS.2009.5161185.

Ashton, Kevin (2009). “That ’Internet of Things’ Thing”. en. In: *RFID Journal*, p. 1. URL: http://www.rfidjournal.com/articles/view?4986.

Barthélemy, M. (2004). “Betweenness centrality in large complex networks”. In: *The European Physical Journal B - Condensed Matter* 38.2, pp. 163–168. ISSN: 1434-6028, 1434-6036. DOI: 10.1140/epjb/e2004-00111-4. URL: http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1140/epjb/e2004-00111-4.

Berde, Pankaj et al. (2014). “ONOS: Towards an Open, Distributed SDN OS”. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN ’14. New York, NY, USA: ACM, pp. 1–6. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620744. URL: http://doi.acm.org/10.1145/2620728.2620744.

Beyene, Million Aregawi (2017). *Evaluation of SDN in Small Wireless-capable and Resource-constrained Devices*. Tech. rep. Norwegian University of Science, Technology Department of Information Security, and Communication Technology.

Buratti, C. et al. (2016). “Testing Protocols for the Internet of Things on the EuWIn Platform”. In: *IEEE Internet of Things Journal* 3.1, pp. 124–133. ISSN: 2327-4662. DOI: 10.1109/JIOT.2015.2462030.

Dio, P. Di et al. (2016). “Exploiting state information to support QoS in Software-Defined WSNs”. In: *2016 Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, pp. 1–7. DOI: 10.1109/MedHocNet.2016.7528421.

Dunkels, A., B. Gronvall, and T. Voigt (2004). “Contiki - a lightweight and flexible operating system for tiny networked sensors”. In: *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462. DOI: 10.1109/LCN.2004.38.

El-Darymli, Khalid and Mohamed Hossam Ahmed (2012). “Wireless Sensor Network Testbeds: A Survey”. In: *Wireless Sensor Networks and Energy Efficiency: Protocols, Routing and Management*, pp. 148–205. ISBN: 978-1-4666-0101-7. DOI: 10.4018/978-1-4666-0101-7.ch007.

Galluccio, L. et al. (2015a). "Reprogramming Wireless Sensor Networks by using SDN-WISE: A hands-on demo". In: *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 19–20. DOI: 10.1109/INFCOMW.2015.7179322.

– (2015b). "SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks". In: *2015 IEEE Conference on Computer Communications (INFOCOM)*, pp. 513–521. DOI: 10.1109/INFOCOM.2015.7218418.

GraphStream - A Dynamic Graph Library. URL: http://graphstream-project.org/.

Guan, Y. et al. (2017). "An open virtual neighbourhood network to connect IoT infrastructures and smart objects #x2014; Vicinity: IoT enables interoperability as a service". In: *2017 Global Internet of Things Summit (GIoTS)*, pp. 1–6. DOI: 10.1109/GIOTS.2017.8016233.

Hurni, Philipp et al. (2011). "TARWIS: A Testbed Management Architecture for Wireless Sensor Network Testbeds". In: *Proceedings of the 7th International Conference on Network and Services Management*. CNSM '11. Laxenburg, Austria, Austria: International Federation for Information Processing, pp. 320–323. ISBN: 978-3-901882-44-9. URL: http://dl.acm.org/citation.cfm?id=2147671.2147726.

Jelasity, Márk (2013). "Gossip-based Protocols for Large-scale Distributed Systems". PhD Thesis. szte.

Kobo, H. I., A. M. Abu-Mahfouz, and G. P. Hancke (2017). "A Survey on Software-Defined Wireless Sensor Networks: Challenges and Design Requirements". In: *IEEE Access* 5, pp. 1872–1899. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2666200.

Kreutz, Diego et al. (2015). "Software-Defined Networking: A Comprehensive Survey". en. In: *Proceedings of the IEEE* 103.1, pp. 14–76. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2014.2371999. URL: http://ieeexplore.ieee.org/document/6994333/.

LoRa Alliance, ed. (2015). *Technical Overview of LoRa and LoRaWAN*. en.

Maven – Introduction. URL: https://maven.apache.org/what-is-maven.html.

Open Networking Foundation (2012). *Software-Defined Networking: The New Norm for Networks*.

Osterlind, F. et al. (2006). "Cross-Level Sensor Network Simulation with COOJA". In: *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, pp. 641–648. DOI: 10.1109/LCN.2006.322172.

Riley, George F. and Thomas R. Henderson (2010). "The ns-3 Network Simulator". en. In: *Modeling and Tools for Network Simulation*. Springer, Berlin, Heidelberg, pp. 15–34. ISBN: 978-3-642-12330-6 978-3-642-12331-3. DOI: 10.1007/978-3-642-12331-3_2. URL: https://link.springer.com/chapter/10.1007/978-3-642-12331-3_2.

Schärer, Jakob, Severin David Zumbrunn, and Torsten Braun (2017). "Universal Large Scale Sensor Network". eng. In: *Schärer, Jakob; Zumbrunn, Severin David; Braun, Torsten (18 September 2017). Universal Large Scale Sensor Network. In: Trajanov, D.; Bakeva, V. (eds.) 9th International Conference on ICT Innovations 2017. Skopje, Macedonia. 18 - 23 Sep 2017. 10.1007/978-3-319-67597-8_8 <http://dx.doi.org/10.1007/978-3-319-67597-8_8>*. Ed. by D. Trajanov and V. Bakeva. Vol. 778. Skopje, Macedonia, pp. 79–88. ISBN: 978-3-319-67597-8. DOI: info:doi:10.1007/978-3-319-67597-8_8. URL: https://boris.unibe.ch/105583/.

SDN-Wise. URL: http://sdn-wise.dieei.unict.it/.

Varga, Andras (2010). "OMNeT++". en. In: *Modeling and Tools for Network Simulation*. Springer, Berlin, Heidelberg, pp. 35–59. ISBN: 978-3-642-12330-6 978-3-642-12331-3. DOI: 10.1007/978-3-642-12331-3_3. URL: https://link.springer.com/chapter/10.1007/978-3-642-12331-3_3.

Winter, T. et al. (2012). *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. en. Tech. rep. RFC6550. RFC Editor. DOI: 10.17487/rfc6550. URL: https://www.rfc-editor.org/info/rfc6550.

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname:

Matrikelnummer:

Studiengang:

Bachelor ☐ Master ☐ Dissertation ☐

Titel der Arbeit:

LeiterIn der Arbeit:

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist. Ich gewähre hiermit Einsicht in diese Arbeit.

Ort/Datum

Unterschrift