A comparative study of route update strategies in SDN based WSN

Bachelor Thesis

Dave Meier

Philosophisch-naturwissenschaftliche Fakultät

der Universität Bern

Tutor

Jakob Schärer

Supervisor

Prof. Dr. T. Braun

January 2020

Abstract

Managing large-scale wireless sensor networks (WSN) is a non-trivial task due to the diversity of sensor node types and complex network topologies. With software-defined networking (SDN), the management of these complex network topologies can be simplified by the global view of the SDN controller. In order to send packets from one node to another, a route needs to be installed between these two nodes. However, it is possible that some sensor nodes on the route fail or are moved away, and so the route is interrupted. To fix interrupted routes, the current implementation of [Schärer et al., 2019] drops all existing routes in a certain interval. After dropping a route, the next time a sensor node wants to send a packet to another node, it sends a request to the controller and then the route gets installed again. This approach has two major drawbacks. First, routes that never change are also dropped with every interval. This leads to an overhead of control packets. Second, routes that change more often must wait up to the given interval until they are updated. This thesis focuses on improving the current implementation to reduce the control packet overhead and simultaneously reduce the time for a route to get updated. For this, two new update strategies are implemented and compared together with the current approach.

Contents

List of Figures

List of Abbreviations

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
SDN	**S**oftware-**D**efined **N**etworking
SDWSN	**S**oftware-**D**efined **W**ireless **S**ensor **N**etwork
WSN	**W**ireless **S**ensor **N**etwork

1 Introduction

This thesis focuses on route update strategies in wireless sensor networks (WSNs) that are based on software-defined networking (SDN). For this, it uses SDNWise, a framework for SDN based WSNs. SDNWise is a direct derivation of Openflow. It uses reactive routing to install routes in the network. In case of a dynamic network, in which nodes might fail or might be moved, it is necessary to update the routes.

1.1 Motivation

The current implementation to update routes in the WSN is very simple and limited: Every 60 seconds, each node drops its flow table and so all routes in the network are dropped. The next time a node needs to send a packet to another node, it has to send a request to the controller again to install a new route.

This approach does not consider how dynamic a route is. It can happen that some routes are very stable and almost never need an update, while other routes need updates more frequently. In the first case, the disadvantage is a certain control packet overhead. This leads to a higher power consumption and to a network overload. In the second case, the disadvantage is a higher packet loss rate because a route may break and then every packet is lost until the route is updated.

1.2 Objectives

The goal of this thesis is to find two alternative update strategies, compare them with the current implementation and then find the most efficient one.
The goals are:

- Defining which properties represent a good update strategy
- Defining and implementing two update strategies
- Comparing all strategies and find the best one

1.3 Thesis Contributions

The main contribution of this thesis are two new route update strategies for SDN based WSNs. The first one is a trickle timer, which replaces the current timer on the nodes in the network. The second one is an implementation on the controller, which detects necessary route updates and sends them to the nodes.

Another contribution is the comparison of the different route update strategies to find the best one. For this, the route update strategies were executed on different WSNs and then metrics were calculated.

A third contribution is an extension for the neighbor table of the nodes in SDNWise. The current neighbor table was not designed for dynamic networks. The nodes did not remove the corresponding entry after a neighbor had disappeared. Because of this, the controller did not receive correct reports from the nodes. This resulted in a false global view of the controller and so in wrongly computed routes. This thesis fixes this issue by deleting a neighbor from the neighbor table after the neighbor did not send a beacon for three times in a row.

A further contribution is an update of the mobility plugin to the newest version of Contiki. Contiki is an open source operating system for the Internet of Things.

1.4 Thesis Structure

This thesis consists of 5 chapters. In chapter 1, an introduction to the subject of this thesis is given. In chapter 2, the relevant background is explained. Chapter 2 also shows the related work. In chapter 3, the two main approaches are presented and explained. Chapter 4 contains the measurements which were executed to find the best update strategy. Chapter 5 contains a conclusion and discusses future work.

2 Background and Literature

This section gives an overview of the related works and explains some of the used techniques.

2.1 Related Work

There are several papers and projects related to this thesis.

[Galluccio et al., 2015] proposed a SDN framework for WSN which is used in this thesis. The framework is called SDN-Wise and is explained in more detail in section 2.4.

[Schärer et al., 2019] proposed a software-defined wireless sensor network testbed. It is a real-world testbed consisting of sensor nodes which are distributed over two buildings. The integration of the SDN-Wise framework is done with TARWIS [Hurni et al., 2011]. This is a testbed management architecture for real-world WSN testbeds.

[Zumbrunn, 2019] is an SDN-framework for securing IoT networks using machine learning. It supports automatic attack recognition, attacker identification and attack mitigation. It is based on anomaly detection in combination with machine learning classification and software-defined networking. The attack recognition and mitigation is not just designed for one certain attack, but rather a more general approach against many different attacks.

[Schärer et al., 2018] is a dynamic traffic aware routing protocol (DTARP) for wireless sensor networks. It is based on SDN and so it has access to global topology information for routing decisions. For the routing decisions, DTARP uses up-to-date traffic information and topology metrics. Its goal is to increase the overall energy efficiency.

The mobility plugin ([Mobility-Plugin]) was used to simulate the dynamics of the network.

2.2 Wireless Sensor Networks (WSN)

A Wireless Sensor Network (WSN) includes a set of spatially dispersed sensors, which can send and receive information via radio. They can be used to measure environmental conditions like temperature, sound, humidity, etc. To gather the recorded information from the sensors, it would be a tedious task to go to every single node and take the recorded information. This is an important reason why we connect the sensor nodes to a network. The network can then be used to send the information to one or more central places. These central places are linked to the network via so-called sink nodes. Sink nodes are just like normal sensor nodes, with the difference that they are usually linked to an outside device or network.

2.3 Software-Defined Networking (SDN)

Software-Defined Networking (SDN) is a networking concept that centralizes network management. In order to do so, it separates the forwarding process of network packets (data plane) from the routing process (control plane).
The data plane is the actual network and consists of the network nodes. The network nodes send their local information as a report to the control plane. The control plane consists of one or more controllers. These controllers take the report from the network nodes and create a global view of the whole network. With the global view, the controllers can compute routes and send them to the data plane.

2.4 SDN-Wise

SDN-Wise ([Galluccio et al., 2015]) 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]). SDN-Wise is directly derived from OpenFlow. A major advantage of SDN-Wise compared to Openflow is that it has a flexible definition of flow table rules. Each flow table entry

consists of a window and an action. If a packet matches the window of a flow table entry, the corresponding action is executed. The action can either be forwarding, dropping or modifying the packet.

2.4.1 System Overview

This section gives a brief overview of all components used in SDN-Wise.

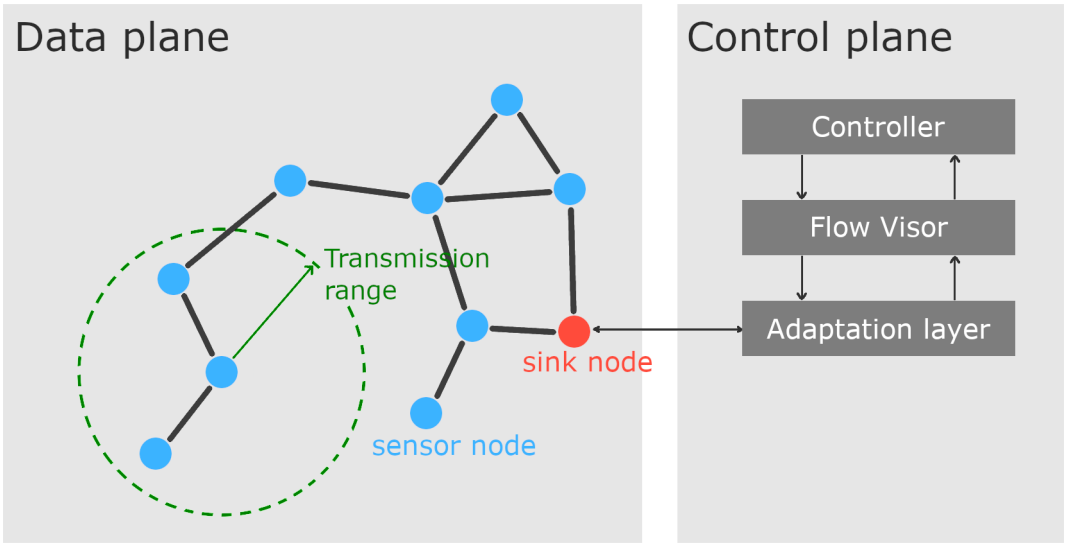


Figure 2.1: Overview of SDN in WSN

Figure 2.1 shows an example of a network using SDN-Wise. On the data plane, there are sensor nodes and a sink node. It is also possible to have multiple sink nodes on the data plane. Each sensor node has a limited transmission range, so if it needs to send a packet to the sink node and the sink node is not in its transmission range, other nodes need to forward the packet to the sink node. The sink node is connected to the control plane, which is responsible for managing the network.

The data plane can either be simulated with a software or it can consist of real-world sensor nodes. When simulated nodes are used, the sink node can be connected with the adaptation layer using TCP/IP. Otherwise, when real-world nodes are used, the sink node can be connected via USB.

2.4.2 Sensor Node Firmware

For the sensor nodes, the SDN-Wise framework contains a firmware called SDN-Wise Contiki. SDN-Wise Contiki is based on Contiki OS and is written in C. The most important features of this firmware are:

- Main loop for handling internal messages
- Timer loop for sending beacons every few seconds
- Timer loop for sending reports every few seconds
- Packet handler for handling incoming packets
- Flow table for storing flow table entries as windows and actions
- Neighbor table for storing neighbors

This firmware is installed on every node. For the sink, the firmware uses the same codebase, but there are preprocessor commands in the code to insert or remove code if the firmware is compiled for the sink. For example, only the sink node needs to send a RegProxy packet at the start to report its existence to the controller.

2.4.3 Control Plane Software

The main software of the control plane using SDN-Wise is called SDN-Wise Java. It is a Java application, which consists of three main parts:

- Adaptation layer
- Flow Visor
- Controller

The adaptation layer is responsible for the interface between SDN-Wise Java and the sink node. After receiving a packet from the sink, the adaptation layer forwards it one layer up to the flow visor.

The flow visor is needed when the network uses more than one controller. For every packet coming from the adaptation layer, the flow visor forwards it to an appropriate controller. The flow visor also needs to forward packets from the controller to the adaptation layer.

The controller stores the global view of the network and calculates new routes for the sensor nodes. Its main responsibility is to handle packets coming from the flow visor. After handling a packet, the controller can send packets back to the flow visor.

It is also possible to use another controller called Open Network Operating System (ONOS) instead of the built-in controller. ONOS is a publicly available Network Operating System written in Java ([ONOS]). To communicate with sink nodes using SDN-Wise Contiki as firmware, SDN-Wise Java is needed as intermediary. The flow visor of SDN-Wise Java needs to be configured to forward packets to the ONOS controller instead of the built-in SDN-Wise Java controller. It is also necessary to start the plugin "SDN-WISE Provider" on ONOS in order to setup a connection to SDN-Wise Java ([SDN-Wise Provider]).

2.4.4 Packet Formats

SDN-Wise is an architecture which uses packets to send data between nodes. There are eight different packet types in SDN-Wise:

- Data
- Beacon
- Report
- Request
- Response
- OpenPath
- Config
- RegProxy

Each packet contains a header with the same structure for all packets. The header consists of 10 bytes and the structure is shown in Figure 2.2.

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

Figure 2.2: Header of packets

The entries of the header store the following informations:

LEN The length of the packet

ID The network id

SRC The source address

DST The destination address

TYP The type of the packet

TTL How many hops the packet can take until it gets deleted

NX HOP The address of the next hop

In the following there is a brief summary for each packet type.

Data packet

Header	Payload
10 Bytes	n Bytes

Figure 2.3: Data packet

The Data packet consists of a header with TYP = 0 and a payload which contains the data. It can be used to send application specific data.

Beacon packet

Header	Distance	Battery
10 Bytes	1 Byte	1 Byte

Figure 2.4: Beacon packet

The Beacon packet consists of a header with TYP = 1, a distance and the battery level. Every five seconds, each node sends a Beacon packet to all

nodes which are close enough to receive it. It is used to tell all neighbors the own distance from the sink and the battery level. Each neighbor adds or updates an entry in its neighbor-table after receiving the packet and checks if the sending node provides a better route towards the sink. If so, the neighbor changes the variable which indicates the standard next hop.

Report packet

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 2.5: Report packet

The Report packet with TYP = 2 is used to send local information to the controller. As soon as the node knows the next hop towards the sink, it begins to send Report packets to the sink in a certain interval. This report is used by the controller to generate a global view of the whole network.
The distance and the battery level is equivalent to the Beacon packet. The node also reports its neighbors by sending the number of neighbors and for each neighbor its address and the Received Signal Strenght Indication (RSSI).

Request packet

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

Figure 2.6: Request packet

The Request packet is used when a node has to handle a packet for which it cannot find a matching entry in its flow table. It consists of a header with TYP = 3, the request id and the packet for which there was no flow entry. It might happen that the unmatched packet is too long to fit into one request. If this is the case, the Request packet is split into multiple packets. In order

to support this, every Request packet stores the number of the current part and the number of parts in total. After generating the Request packet, the node sends it to the controller and then it's up to the controller to decide what to do with the Request packet. The controller used in this thesis calculates a route and send it back to the requesting node with an OpenPath packet. It also tries to send the unmatched packet to its destination address shortly after the OpenPath packet.

Response packet

Header	Rule
10 Bytes	n Bytes

Figure 2.7: Response packet

The Response packet can be used by the controller to send flow rules to a single node. It consists of a header with TYP = 4 and the flow rules to install on the destination node. In order to install a whole route, this packet is very inefficient, because the controller would have to send Response packets to every node in the route. To overcome this, there is the OpenPath packet.

OpenPath packet

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

Figure 2.8: OpenPath packet

The OpenPath packet can be used by the controller to install a complete route between two nodes. It consists of a header with TYP = 5, the window size, the windows for the flow entries, and the addresses of the nodes appearing in the route. To install the route, the packet is sent to the first node on the route. The first node adds a flow table entry and then forwards the packet to

the next node on the route. The next node also adds a flow table entry and forwards the packet to the third node on the route, and so on.

Config packet

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

Figure 2.9: Config packet

The Config packet is used by the controller to change the configuration of a node. It consists of a header with TYP = 6 and the id of the configuration that should be changed. It may also contain parameters which can hold any information. In most cases the parameters hold the new value for the configuration.

RegProxy packet

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

Figure 2.10: RegProxy packet

The RegProxy packet is used by the sink node to report its existence to the controller. It consists of a header with TYP = 7, the id and MAC address of the sink, the physical port, the IP address of the sink and the TCP port of the sink.

3 Approaches

This chapter shows the two approaches to improve the current route update strategy. Figure 3.1 below shows a summary of some possible implementations and the resulting route update strategies. On the left side of Figure 3.1, there are implementations that can be made on the nodes. On the right side, there are implementations that can be made on the controller. The different implementations on the left side can be combined with the implementations on the right side. The resulting combinations represent a route update strategy.

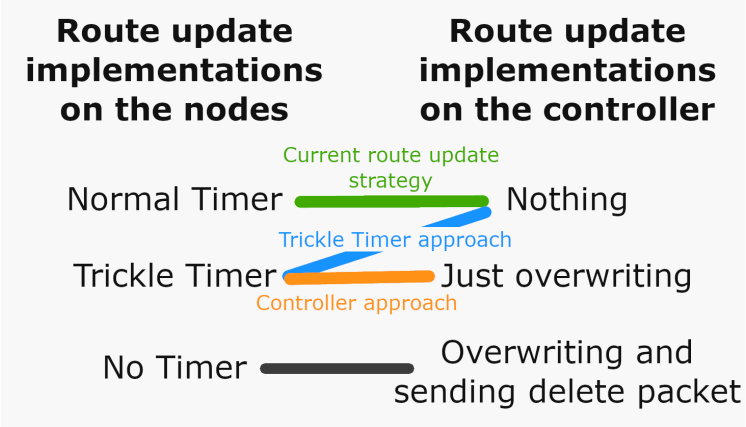


Figure 3.1: Approaches summary

The current implementation for updating routes uses a normal timer on the nodes and no additional implementation on the controller. In Figure 3.1, it is colored green.

The first new route update strategy, the trickle timer approach, is colored blue. It uses a trickle timer on the nodes, which is explained in detail in Section 3.1.

The second new route update strategy, the controller approach, is colored orange. It uses an implementation on the controller that monitors the topology and updates routes immediately. This approach installs an updated route by overwriting existing routing entries. It is possible that some parts of the old route stay in the network, and so it is necessary to continue to use the trickle timer on the nodes.

If the old route could be removed completely, for example by sending delete packets to remove old routing entries, the trickle timer on the nodes would not be necessary. This approach is colored grey and was not implemented and tested in this thesis.

3.1 Trickle Timer

One strategy to improve the current route update algorithm is using a trickle timer. A trickle timer is a timer that increases its interval every time the timer expires. It is possible to reset the interval to its default value after a certain event. The original purpose of a trickle timer is to reduce the control traffic overhead by holding back messages as described in [Lewis et al., 2011]. An improved trickle timer was proposed by [Goyal and Chand, 2018].

For this route update strategy, a trickle timer is used to replace the normal timer that drops the whole flow table every 60 seconds. The main idea is to increase the interval every time the route did not change, and resetting the timer every time the route changed.

3.1.1 Motivation

The trickle timer has the advantage that its interval is adapted to the route changing frequency of the nodes. If a node changes its routes often, the interval will not increase much. If a node has only static routes, the interval will become bigger. By replacing the current timer with the trickle timer, the expected changes are:

- Number of requests from the nodes: will decrease
- Time between sending and receiving a packet: will increase
- Packet loss rate: will increase

Note that the trickle timer has the disadvantage that it increases the maximum of time it takes until a route is updated. This means it may be necessary for a node to wait more than 60 seconds to get an update. If a node in a route fails,

the route will remain incomplete and all packets sent over this route are lost until the trickle timer expires. By increasing the maximum interval time, the packet loss rate will increase if the minimum interval time stays the same. For the time between sending and receiving a packet, it is expected that the time will increase because updating the routes is less frequent. If a better route appears, it takes more time in average until this route is chosen.

3.1.2 Implementation

As already mentioned, the trickle timer replaces the current timer that drops all flow table entries every 60 seconds. So each node has its own trickle timer and if it expires the node drops its flow table.

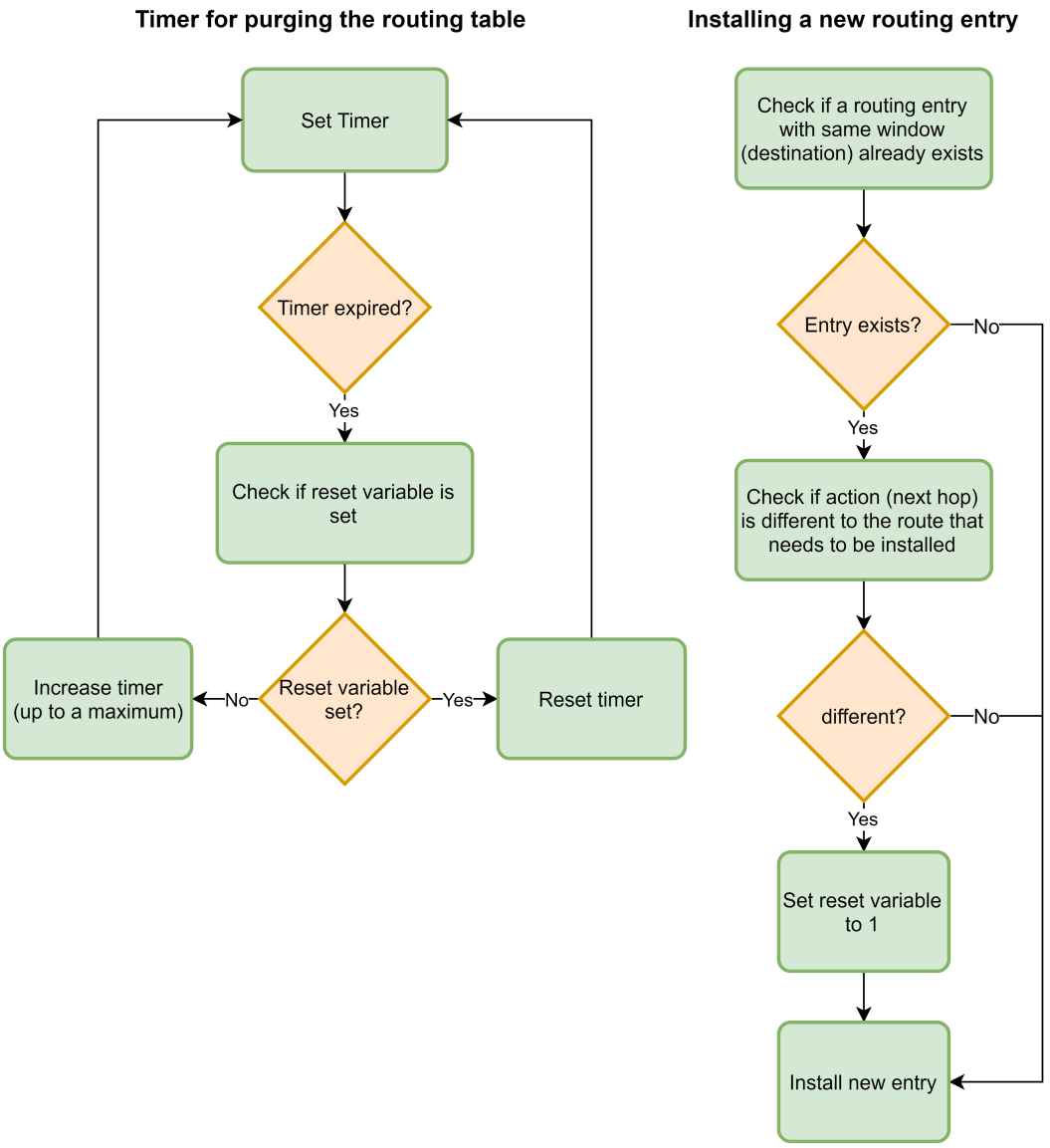


Figure 3.2: Trickle Timer

Figure 3.2 shows on the left side the procedure of setting and resetting the trickle timer. As soon as the timer expires, it checks if the reset variable of the right procedure is set. If this is the case, the interval of the timer is reset to its original value. Otherwise, the timer is increased by the step size. The experiments used as default interval 60 seconds, a step size of 30 seconds and a maximum interval of 180 seconds.

On the right side, Figure 3.2 shows how the reset variable is set. As soon as a new flow table entry needs to be installed, it checks if a routing entry with the same window (destination) already exists. If this is the case and if the action

(next hop) is different as the entry that needs to be installed, the node knows that the route has changed. It sets the reset variable to "1" and the next time the trickle timer expires, its interval is reset to its original value.

Parameters

The trickle timer used in this thesis has the following parameters:

- Default interval
- Maximum interval
- Step size

The default interval is the interval of the timer at the beginning. It is also used for resetting the interval if the route has changed. The maximum interval is a parameter that prevents the interval from becoming too large. The step size is a fixed amount the interval grows every time the route did not change. In [Lewis et al., 2011], the interval was doubled every time an increasing of the timer was necessary. This thesis used a fixed step size to restrain the growing of the interval. A formula of the function is shown below:

$$\text{interval} = \text{default interval} + x * \text{step size}$$

In this formula, x denotes the number of times the timer needed to be increased.

Checking for route changes

It is necessary to keep track of the old flow table entries in order to decide if routes have changed. A first approach would be to copy the whole flow table before dropping it and so keep track of two flow tables. However, this leads to a much higher storage and energy consumption. The disadvantages would surpass the advantages by far.

The approach used in this thesis is more straightforward. Instead of copying the flow table before dropping it, the flow table is not dropped at all. To ensure that old flow table entries are not used anymore for packet handling, an additional flag "deprecated" is added to each flow table entry. This flag indicates that the entry is old and such entries are not used anymore for

packet handling, but only for the trickle timer. This approach increases the storage usage as well by inserting a deprecated flag to each entry and by keeping old entries in the flow table.

3.1.3 Possible Improvements

In this section, some possible improvements for the trickle timer are listed:

Timer per flow table entry

The current trickle timer can only reset the whole flow table instead of single entries. If there is a node with only static routes except for one route that changes more often, the trickle timer will not increase much and there will be no difference to the current approach. A possible solution would be to downgrade the trickle timer to a normal timer and then skipping the deletion for some entries. To implement this, the node would have to store more information per flow table entry. It would be very hard to compare it to the old approach since it would be necessary to compare storage usage with battery usage and decide which one is more important.

Another increasing function

The current trickle timer uses a linear function for increasing its interval. However, it is possible to use any other function to increase the interval. For example, the interval could be doubled each time the timer expires. This would affect the result a lot if the time between "default interval" and "maximum interval" is high. Which function to choose depends on the network structure and network changing behaviour. By changing the step size, the trickle timer can be improved, but the overall concept stays the same and the disadvantages remain.

3.2 Controller Approach

The current route update algorithm can also be improved or even optimized by delegating the process of dropping flow table entries to the controller. Since the controller has a global view of the network, it can check if a node failed or was moved away and then compute the affected routes. After that, it can calculate new routes for each affected route and send them to the network.

3.2.1 Motivation

The current approach as well as the trickle timer approach have the disadvantage that they may drop flow table entries that are still used. The trickle timer is able to delay the dropping of the flow table for some nodes, but at the expense of the packet loss rate. By having a controller with a global view, it is possible to let the controller handle all route updates.

3.2.2 Implementation

To implement the controller approach, there are two major changes of the controller necessary. First, the controller has to check for better routes each time the network structure changes. This is the case if a node reports other neighbors than in the previous report. So, after getting a report, the controller checks if the network structure changed and if this is the case it checks for all affected routes if there is a better route. Second, the controller has to store all routes after computing it in order to find the affected routes after a network structure change.

After finding a better route, it is necessary to send the new route to the nodes. This includes on the one hand the deletion of the old route and on the other hand the installation of the new route. The installation can be done very easily by just sending an OpenPath packet similar to answering requests from the nodes. For the deletion of the routes, there are 3 different possibilities:

- Creating a "Reverse OpenPath packet"
- Creating a "Single Deletion packet"

- Continue to use the trickle timer

Reverse OpenPath Packet

A first idea is to create a new packet type that does the opposite of an OpenPath packet. Instead of going through all nodes on the path and install the route, the packet would go through all nodes and uninstall the route. So, the controller would send a reverse OpenPath packet to the first node on the route. This packet would remove the specific flow table entry and then send the packet to the second node, and so on. Unfortunately, this is not possible since the route will not be complete in most cases. For example, if a node fails and the controller has to find another route because of that, its not possible to remove the old route the same way it was installed.

Single Deletion Packet

Another approach to remove the old route is to create a new packet type which does the opposite of the Response packet. Instead of sending a specific flow table entry, it would remove a specific flow table entry. This packet type could then be used to remove flow table entries on an old route one by one. There are three major drawbacks of this method:

- Many control packets
- Unused routes are not deleted
- Deletion packet may not arrive

The first issue to mention is that in a dynamic network route updates may be necessary very often. This would lead to many control packets, especially if the paths are long. So, this approach may cancel out the advantage it gains with respect to the control packet overhead. Imagine a network with just one route containing 10 nodes which changes every 5 minutes. The current approach sends a Request packet every 60 seconds, i.e. 5 Request packets in 5 minutes. The controller approach with the Single Deletion packet would send every 5 minutes a Single Deletion packet to all nodes, so 10 control packets every 5

minutes. This means in this case the network would even double the amount of control packets to maintain the route.
Another drawback of this approach is that unused routes are kept forever on the nodes. This is difficult to solve, because the controller cannot recognize which routes are no longer needed. So old routes are kept updated by the controller, even if they are no longer needed. The best solution would be if the nodes could drop unused flow table entries themselves as soon as the storage gets full.
The third issue is that a Deletion packet may not arrive at the node, and since the node does not know there was a Deletion packet on the way, it cannot request the packet a second time. From this point on, the controller would have a wrong global view of the installed routes. Even worse, the wrong flow table entry would remain on the node. It would be possible to fix this by sending an Acknowledgement packet from the node to the controller when the packet arrived, but this would increase the control packet overhead even more.

Continue to use the trickle timer

Installing a new route can be done without even uninstalling the old one. This is because when a new flow table entry has to be installed on a node, the node replaces the corresponding old entry (if there is one) with the new entry. The problem is just that the flow table entries that are not contained in the new route are not deleted. This matters only if the node with the wrong entry has to send a packet to the same address. In this case, it sends the packet according to the wrong entry instead of requesting a new route. To solve this problem, a trickle timer can be used. In this case, the advantage of this controller approach remains on reducing the packet loss rate since it updates the routes immediately. This thesis uses the trickle timer in order to support a better comparison between the trickle timer without controller approach and the trickle timer with the controller approach.

4 Measurements and Evaluation

To compare the three update algorithms, each algorithm was tested on a set of different network topologies. For this, a simulation tool called "Cooja" ([Cooja]) was used. Each network was executed for 5 minutes. In total there were 5 different network topologies, on which one simulation per update algorithm was executed. After executing the simulations, different metrics were calculated using a python script.

4.1 Simulation Tool

For the simulation of the network, a virtual machine with Contiki was used. Contiki is an open source operating system for the Internet of Things. It comes with many different tools, one of them is a simulation tool called "Cooja". In this thesis, Cooja was used to simulate the wireless sensor network. Figure 4.1 shows how it looks.

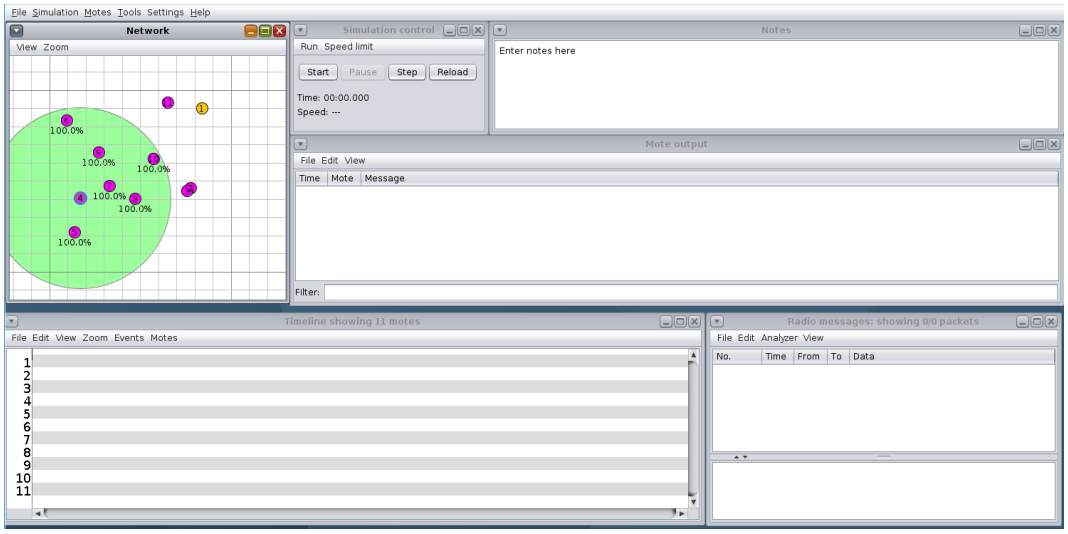


Figure 4.1: Cooja

The sink node in the simulation is colored yellow. It is connected to the controller with a TCP connection. In order to simulate the dynamics of the network, a mobility plugin ([Mobility-Plugin]) was used.

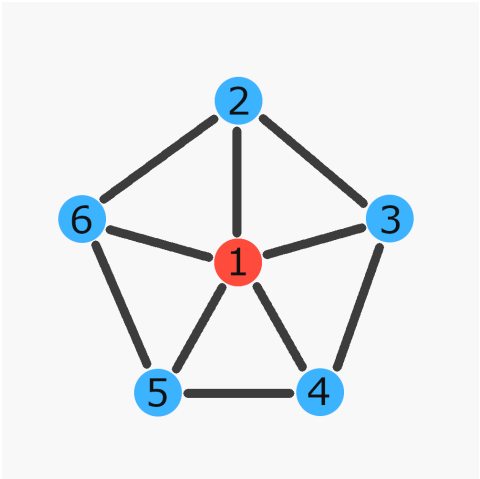
4.1.1 Mobility Plugin

The mobility plugin is able to move nodes from one position to another at a given time. The information for all movements is stored in a text file, in which each line contains the id of the node, the time the movement should occur and the target position. The movements are executed instantly, so when the time is reached, the node immediately goes to the target position.

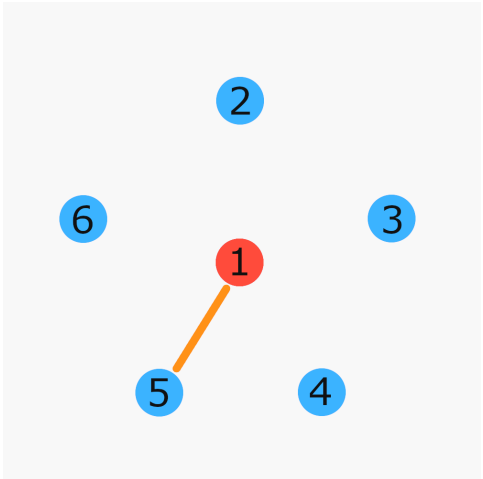
4.2 Topologies

Five different network topologies were used to compare the different update strategies. A graph for each network and the installed routes after 30 seconds are shown below.

Network 1

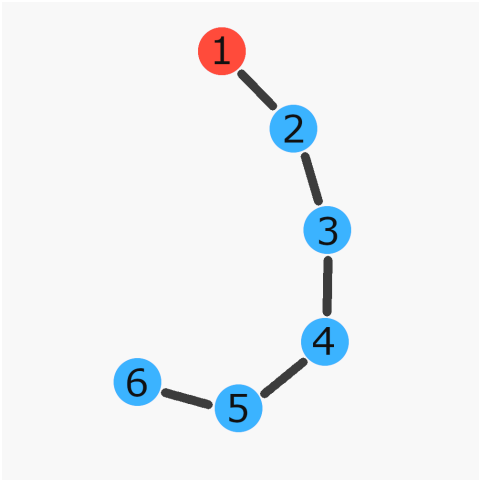


(a) Topology Graph

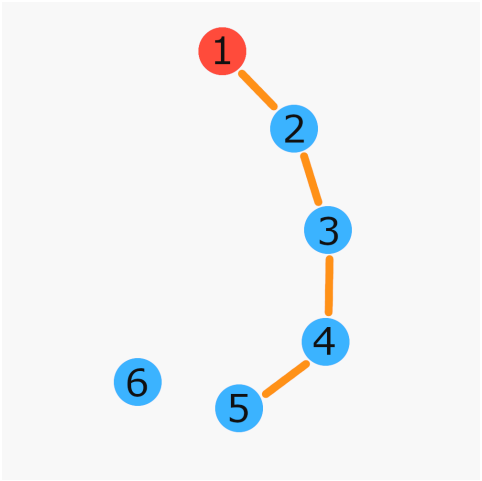


(b) Route 1-5

Network 2

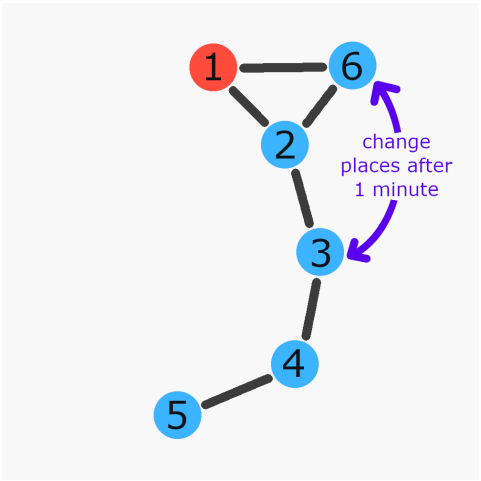


(a) Topology Graph

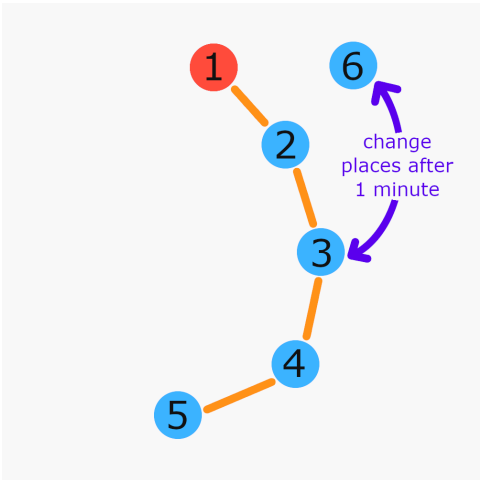


(b) Route 1-5

Network 3

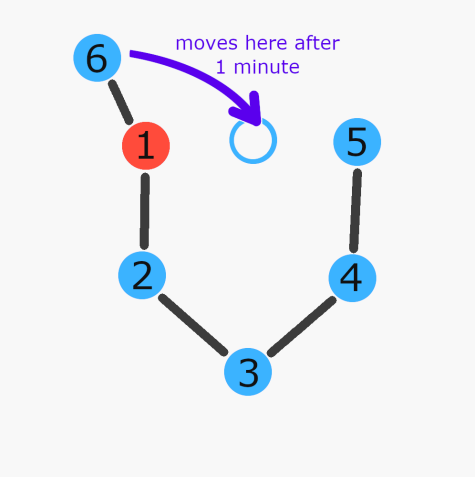


(a) Topology Graph

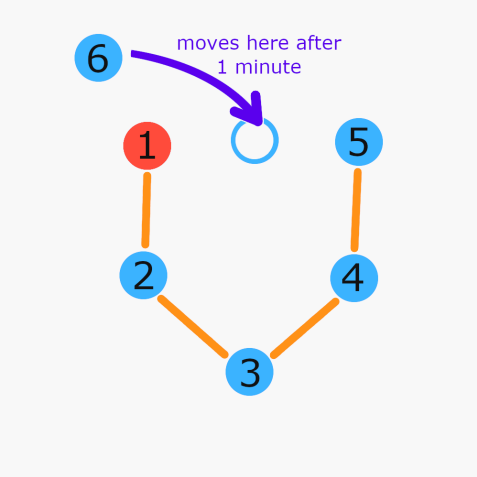


(b) Route 1-5

Network 4

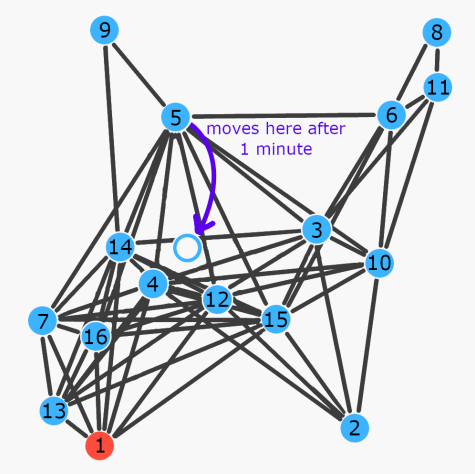


(a) Topology Graph

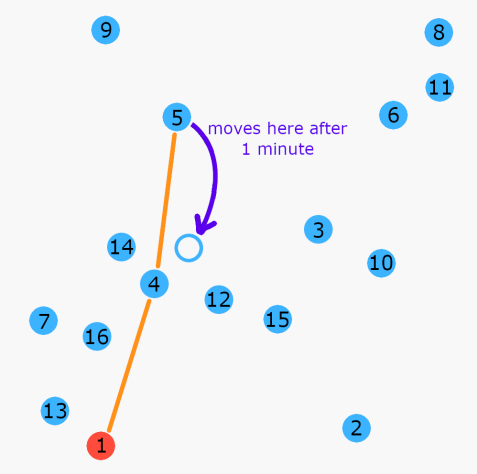


(b) Route 1-5

Network 5



(a) Topology Graph



(b) Route 1-5

Figures (a) show for each network the wireless sensor network topology as a graph. This graph indicates which network nodes are able to communicate with each other. Figures (b) show the route that was installed after the first packet arrived.

4.3 Metrics and Measurements

This section shows the metrics that were used in order to compare the different solutions and their results. Depending on the requirements of the system, one

metric may be more important than others.

For the simulation, node 1 sends a message to node 5 every 10 seconds, starting after 30 seconds to give the controller some time to build the global view of the network. Every simulation was executed for 5 minutes, so in total there were 27 messages sent per simulation.

4.3.1 Number of Request Packets

The first metric used in this thesis is the total number of Request packets per simulation. This metric was used to measure the network traffic overload generated by the Request packets.

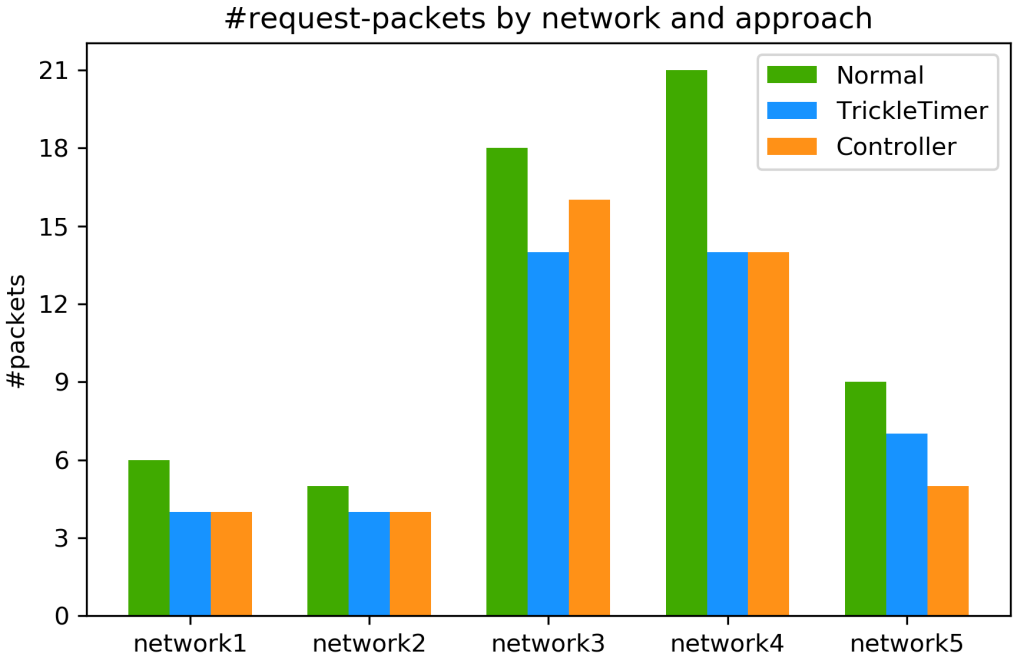


Figure 4.7: Number of requests in total

Figure 4.7 shows to number of Request packets in total. Because the duration of every simulation was 5 minutes, the results are directly comparable. In this figure, it is possible to see that the number of requests was reduced by the trickle timer. The controller approach was also able to reduce the number of Request packets. This result is as expected, because both approaches increase the interval of the timer and so routes stay longer in the network. As long as

a route exists, no Request packets are necessary and so the total number of Request packets decreases.

4.3.2 Average Time Between Sending and Receiving a Packet

The second metric used in this thesis is the average time between sending and receiving a packet. This metric was used to measure "how updated" the routes are in general. The more frequent routes get updated, the more likely it is that the routes are as fast as possible. An assumption for this metric is that the controller always returns the fastest possible route. If the controller computes routes according to other criteria, this metric is not useful.

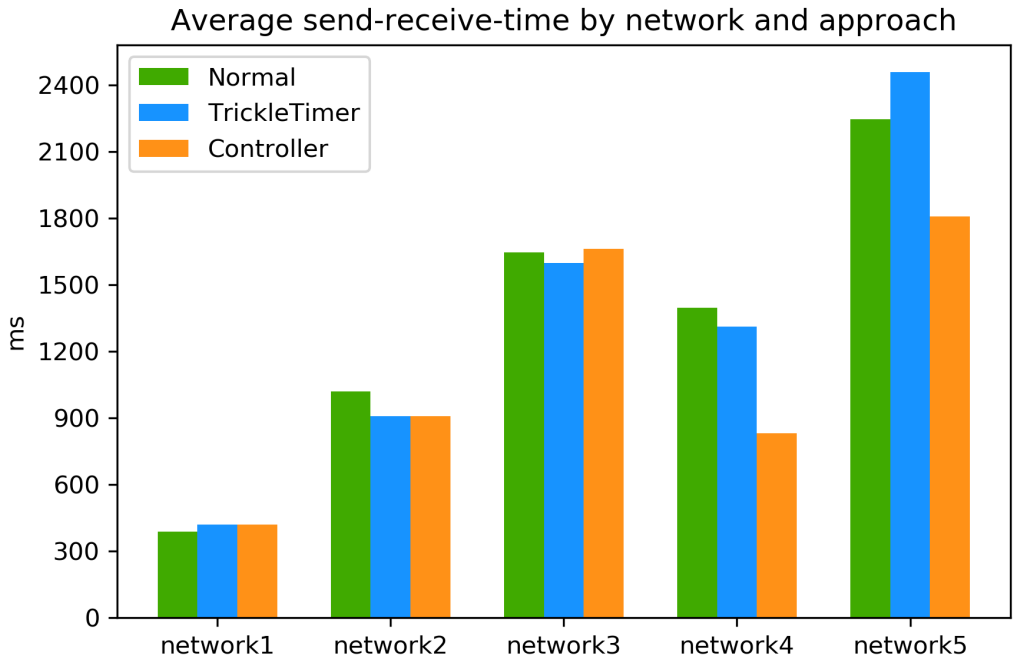


Figure 4.8: Average time between sending and receiving a packet

Figure 4.8 shows the average time between sending and receiving a packet. For networks 1, 2, 3, the time for each approach is almost identical. For network 4, the time of the controller approach is significantly lower than in the other approaches. This is because network 4 is designed to just measure the ability of the update strategy to use a faster route as soon as possible. Network 4 provides a much faster route after 1 minute and so the average time could be

reduced by approaches which immediately start to use the new route.

4.3.3 Packet Loss Rate

The third metric used in this thesis is the packet loss rate. A formula to compute the packet loss rate is shown below.

$$\text{packet loss rate} = \frac{\#\text{lost packets}}{\#\text{sent packets}}$$

This metric was also used to measure "how updated" the routes are in general. The more frequent routes get updated, the less likely it is that a routing entry of a node forwards a packet to a disappeared neighbor and thus the packet gets lost. In the end, this means a smaller packet loss rate for routing strategies that update the routes more frequent.

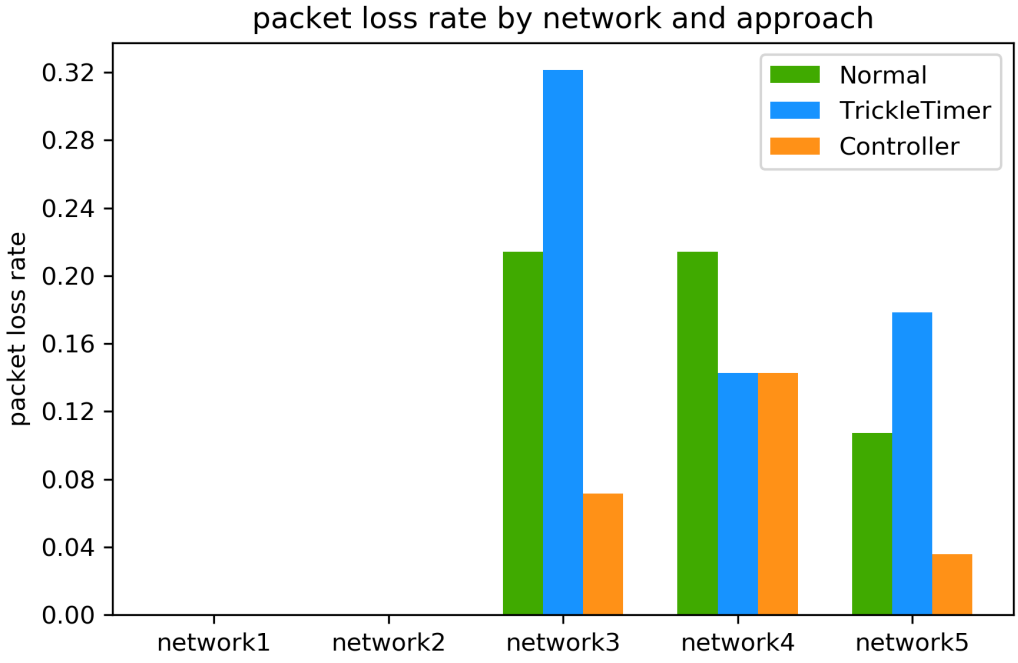


Figure 4.9: Packet Loss Rate

Figure 4.9 shows the packet loss rate of the different approaches. Network 1 and network 2 did not have any lost packages because they are static and because the measurements were executed in a simulation instead of a real world testbed. For network 3, the packet loss rate of the trickle timer is higher than

the current implementation. On the other hand, the packet loss rate of the controller approach is less than the current approach, because it updates the routes immediately. For network 4, the packet loss rate of all approaches is not as expected, because the installed route was never disrupted. So the packet loss rate should be 0 for all approaches in network 4. For network 5, the packet loss rate is similar to network 3.

5 Discussion

In this thesis, two new update strategies were designed and implemented. The first one was an approach to replace the current timer with a trickle timer. This update strategy could reduce the network traffic caused by the Request packets. However, it also increased the maximum time until a route is updated. This lead to many lost packages. The second update strategy resolves this problem by an additional implementation at the controller. This update strategy immediately updates the affected routes after the topology has changed. Because of that, interrupted routes were updated immediately and so it was not necessary to wait for the timer to expire.

5.1 Conclusion

The results of the measurements matched the expectations we had for the update strategies. The trickle timer could reduce the network traffic caused by the Request packets. It also increased the packet loss rate as expected. The controller approach has managed to reduce the packet loss rate while keeping the network traffic as low as with the trickle timer. So there is a clear advantage of the controller approach compared to the current implementation and to the trickle timer. It improves the network traffic and the packet loss rate. The controller approach is also able to reduce the average time between sending and receiving a packet in some cases.

5.2 Future Work

In the current implementation of the controller approach, the controller does not delete the old route. This makes it necessary to continue to use the trickle timer on the nodes. It would be possible to design a new packet type to delete flow entries on a node and then send a packet of this type to delete the old route. However, it might be possible that this packet does not arrive, and so it would be necessary to ensure this packet arrives at the node. An option might be to use Acknowledgement packets, but this could lead to further issues

since it may happen that a link works only in one direction. Additionally, the Delete packets together with the Acknowledgement packets might increase the network traffic even more than the Request packets. So it would be necessary to compare the traffic overload generated by the Request packets with the traffic overload generated by the Delete and Acknowledgement packets.

References

Cooja. Cooja. https://anrg.usc.edu/contiki/index.php/Cooja_Simulator. Accessed: 2020-01-25.

L. Galluccio, S. Milardo, G. Morabito, and S. Palazzo. Sdn-wise: Design, prototyping and experimentation of a stateful sdn solution for wireless sensor networks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 513–521. IEEE, 2015.

S. Goyal and T. Chand. Improved trickle algorithm for routing protocol for low power and lossy networks. *IEEE Sensors Journal*, 18(5):2178–2183, March 2018. ISSN 2379-9153. doi: 10.1109/JSEN.2017.2787584.

P. Hurni, M. Anwander, G. Wagenknecht, T. Staub, and T. Braun. Tarwis — a testbed management architecture for wireless sensor network testbeds. pages 1–4, 01 2011. doi: 10.1109/NOMS.2012.6211968.

P. Lewis, T. H. Clausen, J. Hui, O. Gnawali, and J. Ko. Rfc6206: The trickle algorithm. 03 2011.

Mobility-Plugin. Mobility-Plugin. https://github.com/vaibhav90/Mobilty_Interference_Plugin_Patch_Contiki2.7. Accessed: 2019-12-02.

ONOS. ONOS. https://onosproject.org/. Accessed: 2020-01-25.

J. Schärer, Z. Zhao, and T. Braun. Dtarp: A dynamic traffic aware routing protocol for wireless sensor networks. In *Proceedings of the 7th International Workshop on Real-World Embedded Wireless Systems and Networks*, RealWSN'18, pages 49–54, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6048-7. doi: 10.1145/3277883.3277885. URL http://doi.acm.org/10.1145/3277883.3277885.

J. Schärer, Z. Zhao, J. Carrera, S. Zumbrunn, and T. Braun. Sdnwisebed: A software-defined wsn testbed. In M. R. Palattella, S. Scanzio, and S. Co-

leri Ergen, editors, *Ad-Hoc, Mobile, and Wireless Networks*, pages 317–329, Cham, 2019. Springer International Publishing. ISBN 978-3-030-31831-4.

SDN-Wise. SDN-WISE. `https://sbnwiselab.github.io/`. Accessed: 2019-10-25.

SDN-Wise Provider. SDN-Wise Provider. `https://github.com/sdnwiselab/onos/tree/onos-sdn-wise-1.10/providers/sdnwise`. Accessed: 2020-01-25.

S. Zumbrunn. Sdn-wise anti-attack. Master's thesis, University of Bern, 2019.

Ich erkläre hiermit, dass ich diese Arbeit selbstständig 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 o des Gesetzes vom 5. September 1996 über die Universität zum Entzug des aufgrund dieser Arbeit verliehenen Titels berechtigt ist.

Datum: 24.02.2020 Unterschrift: Dave Meier