Energy Efficient Determinism in WSN through Reverse Packet Elimination

Fredrik Kvist

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Department of Technology Systems
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

Energy Efficient Determinism in WSN through Reverse Packet Elimination

Fredrik Kvist

Energy Efficient Determinism in WSN through Reverse Packet Elimination

http://www.duo.uio.no/

Abstract

In recent years, wired industrial networks has shifted towards Wireless Sensor Networks (WSN). Utilizing WSNs attract interest by various industries as it can provide real time measurements in a cost effective way. Next, with the emerging Industrial Internet of Things (IIoT) WSNs can connect to the Internet. With the creation of Deterministic Networking Group (DetNet), research have been aimed at giving WSNs deterministic capabilities.
In this thesis, a novel Reverse Packet Elimination (RPE) algorithm was implemented at IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) stack with intent to increase reliability without increasing energy consumption significantly. RPE was partially investigated analytically, but mainly through the 6TiSCH Simulator.
Results of introducing RPE revealed that reliability increased with 10.5%, average latency decreased with 27% and lowest node life was increased with 1.5% over links with 70% quality. However, average energy consumption in the network increased with 19.8% compared to not utilizing packet replication (e.g single point of failure).

Contents

List of Tables

List of Figures

Listings

Preface

This thesis concludes my Master's degree at the Department of Technology Systems at the University of Oslo. This thesis was done as part of the collaboration between University of Oslo and Western Norway University of Applied Sciences. Special thanks to PhD Candidate Andreas Urke and my supervisor Professor Knut Øvsthus for excellent counseling. Lastly, thanks to fellow students Eirik Kjevik and Mathias Utgård for their contribution to discussions.

Chapter 1

Introduction

1.1 Background

Currently, Internet of Things (IoT) is emerging with an expectation of billions of connected devices to the Internet [5]. What is more, these devices are utilizing low-power wireless communication technologies enabling long battery life and reliable end-to-end transmissions [10]. With this development, industries are shifting towards Industrial Internet of Things (IIoT) through Wireless Sensor Networks (WSN). Utilizing WSN decreases installation cost as compared to infrastructures for wired sensor networks [11]. For instance, cost of drawing cables across an industrial plant can run from $100s/ft to $1000s/ft [12]. The last decade, prominent solutions such as WirelessHART [13] and ISA100.11a [14] has emerged for Industrial WSN. However, these technologies do not support Internet Protocol (IP) traffic by design and in 2015 an amendment to the Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 standard was released with Time-Slotted Channel Hopping (TSCH). TSCH has proven to increase reliability with deployments demonstrating 99.999% delivery ratio according to [15]. 6TiSCH working group was started by Internet Engineering Task Force (IETF) [16] with intent to enable IP traffic for IIoT utilizing TSCH. In addition, 6TiSCH shall follow IETF Deterministic Network Working Group (DetNet) architecture [17], aiming to standardize layer 3 traffic in terms of networks requiring deterministic capabilities.

1.2 Motivation

In certain industrial networks, a feedback loop is utilized to control a system. The feedback loop is compensated with a control variable as seen in Figure 1.1.

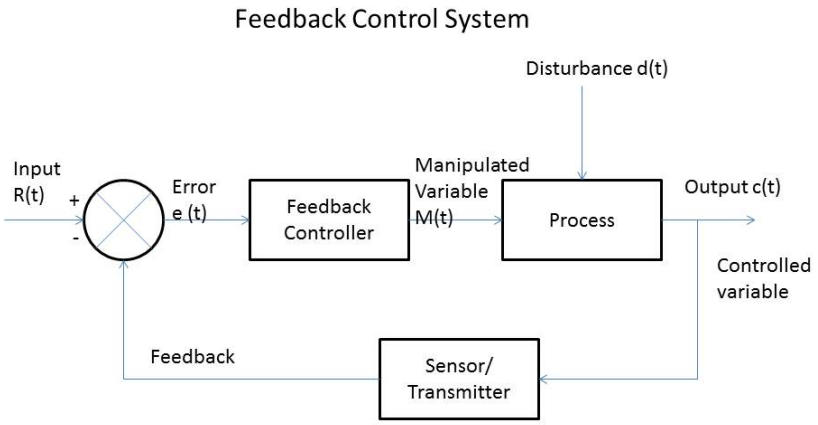


Figure 1.1: Illustration of a control system with a feedback loop retrieved from [4]

This feedback loop is updated periodically and the frequency is determined by the system. In some control systems, the control variable is event triggered, and in others a time based sampling period is implemented [18]. In this thesis, a time based sampling period is assumed, illustrated in Figure 1.2. Hence, a node must transmit a data to ensure control variable is delivered within a maximum latency satisfying the feedback loop.

Figure 1.2: Illustration of a periodically updated control variable

A technique to ensure the control variable delivery is packet replication. With packet replication, packets are duplicated and sent along two disjoint paths. When arriving at their destination they are discarded if a replicated packet was received earlier. In essence, packet replication creates redundancy for data traffic, as they do not have a single point of failure. This is seen in Figure 1.3 where source has two disjoint paths to destination. In short, this increases reliability[17][16] with the drawback of an increase in resource utilization as more packets traverse the network. For example, the control variable illustrated in Figure 1.1 can be replicated along two paths to increase the probability of reception. Packet replication

technique exists in wired networks and in Ethernet; Parallel Redundancy Protocol (PRP) and High-Availability Seamless Redundancy (HSR) can be utilized [19].

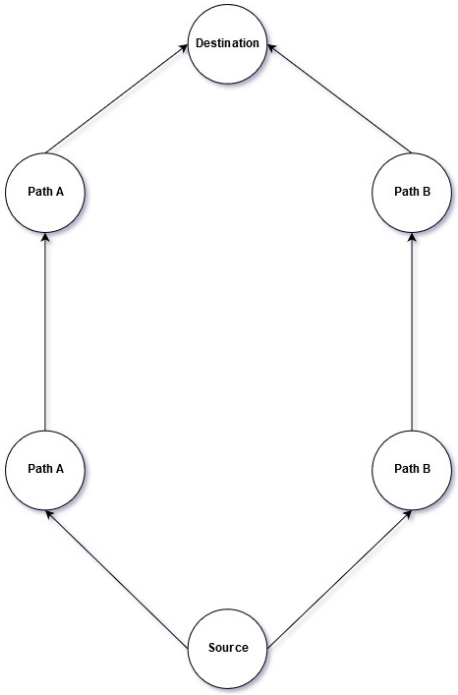


Figure 1.3: Illustration of redundancy to destination from source with two paths

On the other hand, with wireless networks the probability of packet loss increases compared to wired due to interference and channel disturbance. Hence, 6TiSCH recommend packet replication in its architecture to increase reliability [16]. In a wireless network, this means a higher energy consumption [5] and a reduced network lifetime. Although, reliability is increased significantly, justifying the extra energy consumption [5][6][20]. However, any reduction in energy consumption is greatly desired as nodes are battery driven.

1.3 Objective

Objective of this thesis is to investigate whether a Reverse Packet Elimination (RPE) algorithm can increase reliability, and decrease energy consumption in WSNs operating over IEEE 802.15.4 standard, specifically the 6TiSCH stack. As opposed to discarding replicated copies at destination, RPE triggers a sink to send a packet down the opposite path to eliminate the other copy going upstream. In

addition, this thesis examine effects of introducing an adjustable replication delay τ in the packet replication mechanism as seen in Figure 1.4. Figure 1.4 illustrates one period within Figure 1.2.

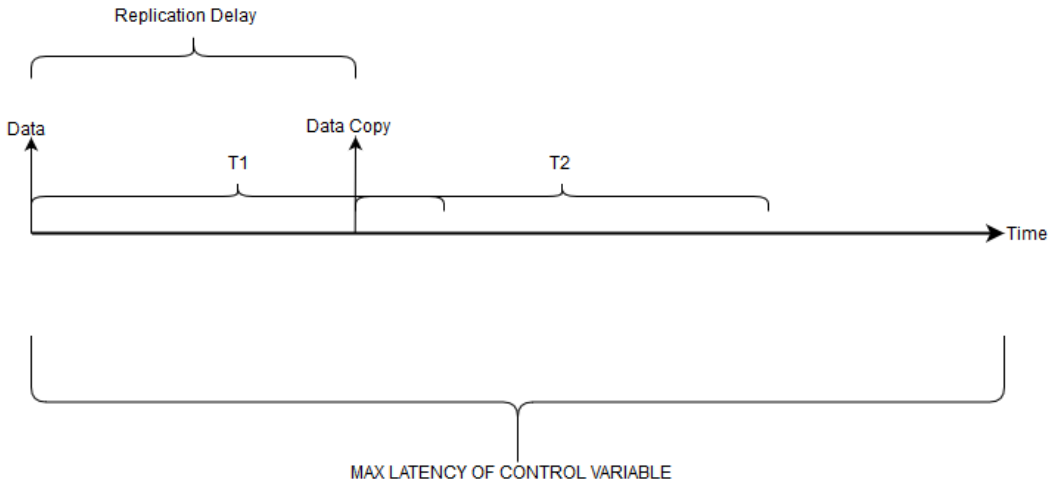


Figure 1.4: Illustration of a periodically updated control variable with a bounded maximum latency

T1 indicates the time it takes to transmit data from source to sink. Next, data copy is withheld by a replication delay, and T2 is ensured to be within maximum latency of the control variable presented in Figure 1.1. When the data is received at sink, it sends a RPE packet to stop the data copy from being transmitted. If the data is lost during transmission, the data copy provides redundancy needed to ensure the upper bound latency.

1.4 Limitations of scope

Researching a full stack architecture across multiple layers encompasses a vast field of technologies. Hence, this thesis is limited to the lower layers of the 6TiSCH stack. Assumptions and design is influenced by Leapfrog Collaboration (LFC) [6][20] presented in Subsection 1.5.1. Moreover, simulations and results are limited to a single topology.

1.5 Related work

To the best of my knowledge, no related work exists in terms of a reverse mechanism or replication delay presented in Section 1.3 and derived in Chapter 3.

Although, in [21] the authors visit the idea of a reverse packet mechanism in Ethernet, with the reversing of packet elimination to free bandwidth upstream. However, no actual discussion or work are provided. Moreover, with the packet replication and elimination techniques utilized in 6TiSCH, energy consumption increases significantly. A rise in energy consumption is expected as the number of packets traversing the network at minimum doubles. This is shown in [5] where the authors implement disjoint paths and send a copy of the packet at each path. In short, source duplicates a packet and sink drop the packet arriving last. They show packet replication decreased packets lost with almost 90%, but energy consumption increases with 86.3%. Figure 1.5 illustrates disjoint paths utilized in [5].

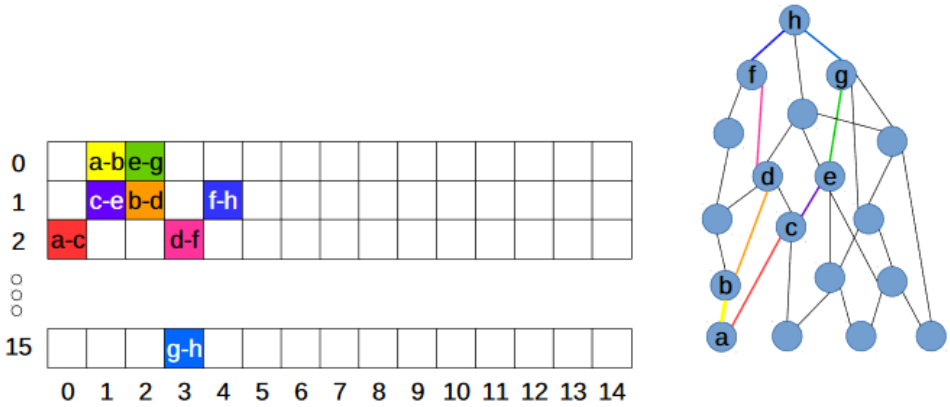


Figure 1.5: Example of disjoint paths retrieved from [5]

1.5.1 Leapfrog Collaboration

Leapfrog Collaboration is designed by some members of the 6TiSCH working group and hence important related work. During this thesis, I have been in contact with the authors of LFC. I reached out with intent of retrieving their implementation for comparison, but they were not willing to share at this point due to further development. LFC utilizes the same principal as [5] by dropping the packet that arrive last. However, LFC does that at each hop as they enable promiscuous overhearing between all motes within sensing range. In short, promiscuous overhearing means all motes are listening. This method requires a Leapfrog Beacon enabling all motes to free the same cell for RX. Opening the same cell allows all motes except the mote transmitting to listen for data. As a result, delay decreases significantly as the packet propagate the best links. Simply put, packets are partially meshed from source towards sink. If no motes are within sensing range, normal retransmission schemes are used. However, as this require all motes in sensing

range to listen for all packets and process them, energy consumption naturally increases compared to sending over a single path. Moreover, Leapfrog Collaboration technique is illustrated in Figure 1.6.

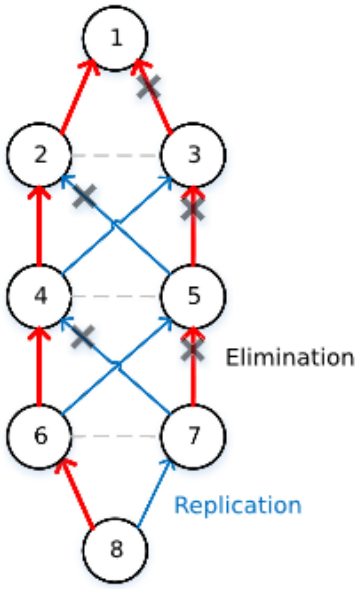


Figure 1.6: Leapfrog Collaboration scheme retrieved from [6]

LFC achieves a minimum end-to-end reliability of 99.1% for all simulations, with lowest Packet Delivery Ratio (PDR) link quality at 70%. However, as mentioned energy consumption increases significantly with worst case drawing 177% more than normal operation. As well as a high end-to-end reliability, LFC reduces average delay 41% compared to standard retransmission schemes [6]. To summarize, results achieved by LFC are impressive, but do require several upstream motes to be within sensing range.

1.6 Thesis structure

Thesis is structured as follows:

- Chapter 1 (this chapter) introduces background information, motivation, objective of this thesis and limitations of the scope. At last, related work and the structure are presented.
- Chapter 2 supplies reader with necessary theory to interpret purposed solution.

- Chapter 3 describes proposed solution in details.
- Chapter 4 present methods utilized to investigate purposed solution. Namely, simulator selection is advocated and a general description provided.
- Chapter 5 implement proposal into the simulator and shows a drawback with the simulator. Then a description of scenarios simulated are provided.
- Chapter 6 starts with theoretical results. Then, simulation results are presented.
- Chapter 7 takes results into discussion and describes traits of each scenario
- Chapter 8 concludes work done. Moreover, a conclusion is drawn based on results and discussions. In addition, ideas for future work are discussed.

Chapter 2

Theory

This chapter presents the theory needed to understand work presented in this thesis.

2.1 Radio frequency propagation

In this section the basics of Radio Frequency (RF) propagation are presented. It is not an in-depth presentation as RF propagation is not subject of analysis in this thesis.

2.1.1 Free space

When modelling aspects of a wireless channel it is common to assume radio waves are propagating through ideal free space without loss. This entails space between receiver and sender is completely free of objects scattering, absorbing or influencing the signal in its path. This means no reflections and the medium itself do not absorb any signal. Transmitting power radiating from an ideal isotropic antenna is attenuated in free space by a factor $L_s(d)$, known as free space loss [22].

$$L_s(d) = \left(\frac{4\pi d}{\lambda}\right)^2 \tag{2.1}$$

d Distance from source
λ Radio wavelength of transmitted signal

An isotropic antenna radiates spherically and uniformly from a point source. It is can also be referred to as an isotropic radiator [22].

2.1.2 Power received

Power received by destination node is calculated by

$$P_R = P_T G_T G_R L_s(d) \tag{2.2}$$

P_t	Output power of transmitter
G_t	Transmitter gain
G_r	Receiver gain
$L_s(d)$	Free space loss

Moreover, receiving node can be influenced by several factors such as multi-path fading, interference from other sources or objects blocking line of sight. Another key point, as IoT nodes are usually located with at a distance greater than two wavelengths, effect of near field communication are not taken into consideration ($d > 2\lambda$).

2.2 IEEE 802.15.4

IEEE 802.15.4 standard was developed to ensure short range communication and is designed to have low-data-rates, long battery life (e.g months or years), low complexity and low hardware cost [10]. Moreover, IEEE 802.15.4 empower simple devices with reliability and robust wireless technology and can be utilized without extensive knowledge of radio technology and communication protocols [1]. Intention is low-duty-cycle communication combined with relatively high data rates. This allows transfers of small blocks of data between devices completed in milliseconds. What is more, IEEE 802.15.4 is limited to the lower layers. Specifically, Medium Access Control (MAC) and physical layer.

2.2.1 Radio frequency parameters

IEEE 802.15.4 utilizes different radio frequency link parameters. These parameters include modulation type, coding, spreading, symbol/bit rate and channel utilization as described in Table 2.1.

Frequency band	868.3 MHz	902-928 MHz	2400-2483.5 MHz
# Of channels	1	10	16
Bandwidth	600 kHz	2000 kHz	5000 kHz
Data rate	20 kbps	40 kbps	250 kbps
Symbol rate	20 ksps	40 ksps	62.5 ksps
Unlicensed geographic usage	Europe	Americas	Worldwide

Table 2.1: IEEE 802.15.4 frequency bands, parameters and channelization retrieved from [1]

2.2.2 Time-slotted channel hopping

IEEE 802.15.4e was introduced in 2012 as an amendment to the MAC protocol utilized in IEEE 802.15.4 standard. The amendment introduced Time-Slotted Channel Hopping (TSCH), which is use of channel hopping to enable high reliability and time synchronization to attain low-power operations. Overall, TSCH decreases link failure due to external interference and multi-path fading. In fact, utilization of channel hopping is key to achieving high reliability with end-to-end packet delivery ratio claimed to be as high as 99.999% [23]. Moreover, as the amendment is located at MAC layer TSCH do not affect the physical layer and can thereby be used with hardware supporting original IEEE 802.15.4. In 2016, this amendment was adopted in IEEE 802.15.4 and is part of the official standard [9]. Furthermore, time synchronization can vary from microseconds to milliseconds and accuracy of the timing impacts power consumption. In fact, the node clock drifts and need to be periodically synchronized. For this reason, data and Acknowledgement (ACK) packets contain information utilized to re-synchronize. It is important for nodes to have synchronized clocks to ensure they are transmitting and listening when they should.

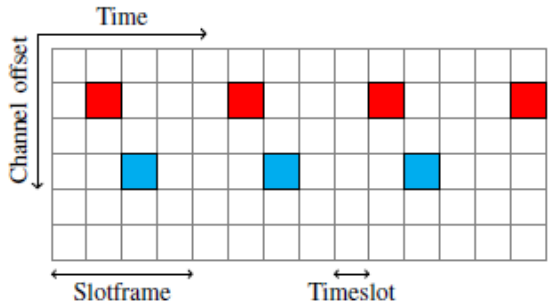


Figure 2.1: TSCH schedule example

In TSCH, time is divided into timeslots as shown in Figure 2.1. Length of a

timeslot is not specified, but a common value is 10 milliseconds[23]. In addition, timeslots are grouped into slotframes commonly consisting of 101 timeslots[23].

2.3 Deterministic-Networking

This section present IETF established Deterministic Network (DetNet) Working Group [24], started with intent to provide guaranteed bandwidth, extremely low loss, and an upper bound on maximum end-to-end latency at layer 3 paths across multiple layer 2 networks. In short, DetNet aims to converge Informational Technology (IT) with Operational Technology (OT) networks [21]. Information provided in this section is retrieved from DetNet Architecture version 10 [17].

Time-Sensitive Networking DetNet working group collaborates with IEEE 802.1 Time-Sensitive Networking (TSN) [25] which is improving on layer 2 operations to ensure deterministic capabilities. A short survey of TSN is presented in Appendix B. In essence, DetNet and TSN define a common architecture for both Layer 2 and Layer 3 to ensure applications requiring determinism can work across both layers. Such applications include engine control systems, general industrial applications, and professional and home audio/video systems.

2.3.1 Primary goals

DetNet Quality of Service (QoS) primary goals are achieving minimum and maximum end-to-end latency, timely delivery, and bound jitter. DetNet should provide a packet loss ratio based on operational states of nodes and links. Ultra high reliability with 99.999% end-to-end packet delivery ratio is desired.

Mechanisms to achieve Quality of Service

DetNet list following mechanisms to achieve QoS:

- Congestion protection
- Service protection
- Explicit routes

Congestion protection

Eliminate congestion loss Primarily DetNet can achieve QoS assurance by eliminating packet loss due to congestion. Next, ensuring each node throughout the network has sufficient buffers, packet loss due to packet drop can be eliminated.

In essence, all DetNet nodes from source to destination need to carefully regulate their output to not exceed the data limit. Moreover, output regulation would in return require time-synchronization of nodes, as a single packet sent ahead of its time could potentially cause a node to reach the resource limit.

Jitter reduction DetNet do not enforce methods to ensure jitter reduction, but simply encourage use of sub-microsecond time synchronization from source to destination.

Service protection

Service protection aims to eliminate or mitigate packet loss due to failure of equipment, random media and/or memory failure. A simple technique to reduce these issues is to send data over multiple disjoint forwarding paths. Multiple paths would be network architecture dependent.

In-order delivery A side effect due to service protection is out-of-order packet delivery. Receiving packets out-of-order may impact the DetNet path. DetNet proposes a maximum allowed out-of-order constraint. If the constraint is set to zero, a path do not tolerate any misordering of packets. In addition, sequencing packets can be done by adding a sequence number or a DetNet time stamp. This association can be inherent in the packet itself or associated with physical properties such as precise time and/or radio channel reception of the packet. Sequencing method should happen once, and close or at to source.

Packet replication and elimination Utilizing a Packet Replication Function (PRF) and replicate packets into multiple paths to the sink increases reliability and is as presented in Section 1.2 desired. Next, when nodes receive a replicated packet, Packet Elimination Function (PEF) drop duplicates based on sequencing information. Output of PEF is always a single packet. Different tactics to remove duplicates can be utilized depending on resources available. For instance, each node from source to destination can perform PEF, but the most common case is to perform PEF close to the edge of the DetNet network. Lastly, when receiving the single packet from PEF nodes should utilize Packet Ordering Function (POF) to re-order packerts received out of order. Order of PRF, PEF and POF are implementation specific. Moreover, packet elimination and replication are passive methods and do not react to, or correct failures.

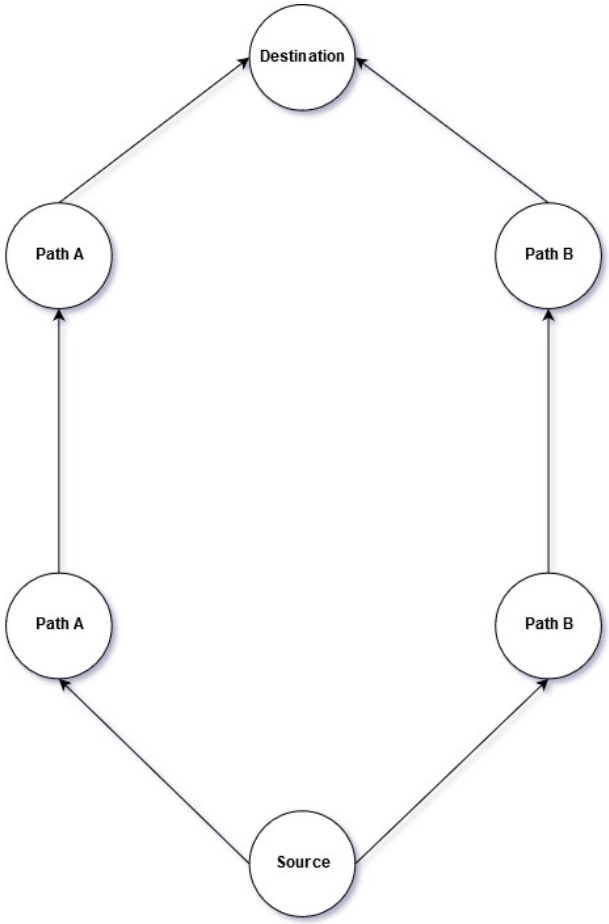


Figure 2.2: Example of disjoint paths from source to sink

Explicit routes

A network topology event in certain parts of the network can briefly interfere with delivery of data. Out-of-order packet delivery can occur due to route changes. Thus, DetNet deploy explicit routes to get advantage of low hop count and assure against very brief losses of connectivity. By utilizing explicit routes, DetNet paths do not change in response to a network topology event. At least not immediately, or in most cases not at all. DetNet states that techniques to establish required explicit routes exist in RSVP-TE [26], Segment Routing [27], Software Defined Networking [28] and IS-IS [29]. Moreover, MPLS TE typically uses explicit routes [30]. Paths set up in Figure 2.2 can be looked upon as explicit routes.

2.3.2 Secondary goals

DetNet has a secondary goal to enable coexistence with normal traffic. Secondary goals are with regards to bandwidth, worst-case latency and transmission opportunities for non-DetNet traffic. Firstly, bandwidth not utilized by a DetNet path can be presented to non-DetNet packets. Secondly, DetNet paths can be scheduled, or shaped, to make sure non-DetNet packets are ensured a upper-bound latency. Lastly, to satisfy the need of non Machine-to-Machine users of the network, DetNet paths should be scheduled in detail. What is more, sufficient transmission opportunities for non-DetNet packets should be taken into considerations.

2.3.3 Stack model

DetNet stack model is designed in two adjacent sub-layers; service sub-layer and transport sub-layer. The DetNet stack model is illustrated in Figure 2.3 [17]. Service sub-layer provides service protection to higher layers through PRF, PRE and POF. Transport sub-layer support the underlying network through congestion protection and explicit routes.

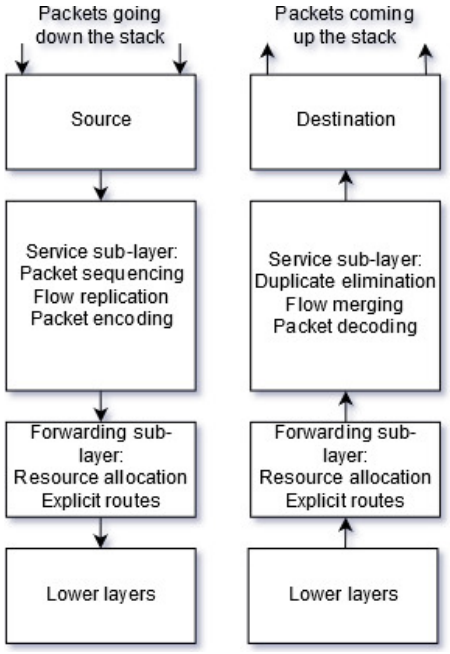


Figure 2.3: Simplified DetNet stack model

Furthermore, sub-layers are application and network specified, meaning all functionality is not required unless asked for. However, networks with critical needs do requires more functionality.

2.3.4 End systems

Data flow between source and destination end systems is known as Application-flow (App-flow). Traffic in an App-flow can have different characteristics such as constant or variable bit rate as well as layer 1-3 encapsulation. These characteristics are used as input when considering resource reservation. However, an end system may not be aware of the DetNet App-flow and it may not contain DetNet specific functionality. End systems are divided it to four categories.

- DetNet unaware: Normal service requiring service proxies.
- DetNet f-aware: Forwarding sub-layer aware system. Do not know about resource allocation but is aware of some TSN functions such as reservation.
- DetNet s-aware: Service sub-layer aware system. Do not know about resource allocation but applies sequence numbers.
- DetNet sf-aware: Full functioning end system. It has DetNet functionality and can be treated as an integral part of the DetNet domain.

Categorization of end systems are illustrated in Figure 2.4.

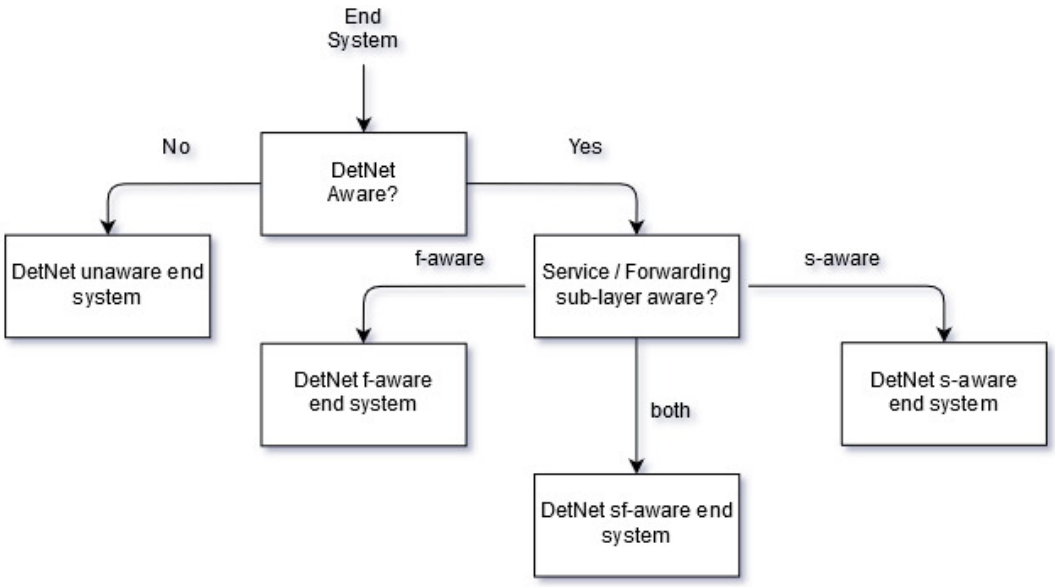


Figure 2.4: Categorization of end systems

2.4 DetNet and 6TiSCH

There are wireless networks supporting real-time QoS such as ISA100.11a and WirelessHART, but they have a drawback that they are incompatible with each other, and do not support IP traffic by design. Furthermore, DetNet firstly has wired networks in mind, but DetNet is to apply 6TiSCH as a wireless network standard for industrial networks [31][16]. Currently 6TiSCH depend on DetNet to define:

- Operation and configuration state for deterministic paths
- End-to-end protocols for deterministic forwarding (IP, tagging)
- Protocol for packet replication and elimination

Achieving these definitions in cooperation with DetNet allow for compatibility with a TSN backbone [31][16].

2.5 IPv6 over the TSCH mode of IEEE 802.15.4e

This section gives a brief introduction to each of the layers in IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) stack [16]. It has standards developed from both IETF and IEEE, which together achieves end-to-end connectivity with deterministic capabilities, low power operation and robustness [12]. Moreover, 6TiSCH stack was created to enable industrial process monitoring and control, and unleash Industrial Internet of Things [32]. 6TiSCH standardization is still in process and as of 2019 not finished [16].

2.5.1 Technical overview

Figure 2.5 illustrates the 6TiSCH protocol stack [16]. Moreover, the 6TiSCH stack encompasses IEEE 802.15.4e known as TSCH, 6LoWPAN, RPL and CoAP to integrate OT with IT. The following subsections describe the stack from the bottom up.

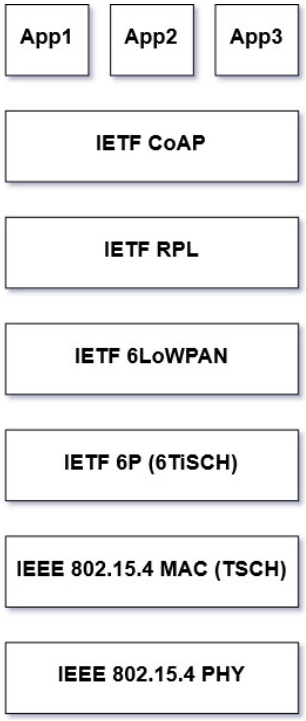


Figure 2.5: 6TiSCH stack model

802.15.4 PHY and MAC

This layer has previously been described in Section 2.2, and in 6TiSCH parameters utilized at link layer are data rate at 250 kb/s, 2.4 GHz frequency and a transmit power in range 0 - 10 dBm. It is worth noting maximum payload size is 127 bytes.

6top

IETF 6top protocol is standardized by the 6TiSCH Working Group [33]. In principle, 6top layer define a distributed scheduling protocol where neighbor nodes negotiate to add or remove one or more cells in the TSCH schedule. Furthermore, each slot in the schedule is a opportunity for neighbours to exchange link layer frames. Slot opportunities repeat in slotframes.

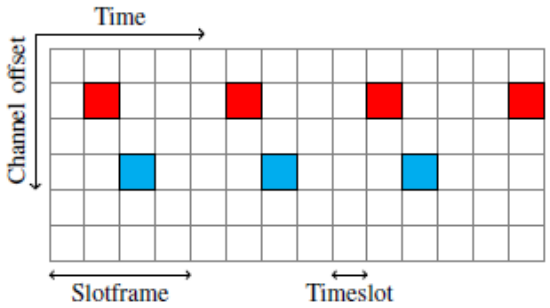


Figure 2.6: TSCH schedule example

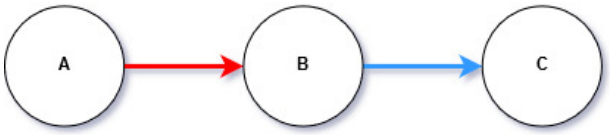


Figure 2.7: Example topology for Figure 2.6

As illustrated in Figure 2.6 motes communicates at different times, slots and channel frequency. In terms of terminology, a cell indicates a timeslot at a given channel offset. A schedule is illustrated for mote B in Figure 2.7, where red is mote A and blue indicates mote B speaking. Note that this is a simple illustration and do not show full complexity of 6top layer. Red and blue indicates a cell at a timeslot. In addition, nodes are referred to as motes in 6TiSCH. Hence, from this point forward all nodes are referred to as motes.

6LoWPAN

6LoWPAN [34] defines a method of fitting Internet Protocol version 6 (IPv6) packets into shorter IEEE 802.15.4 frames. This method works by two main mechanisms. Firstly, adding rules for shortening the IPv6 header:

1. Removing fields that are not needed
2. Removing fields that always has the same content
3. Compressing IPv6 addresses by deriving them from link layer addresses

Secondly, defining fragmentation rules so several IEEE 802.15.4 packets can make up one IPv6 packet. By doing these steps, IPv6 packets as long as 1280 bytes fit into IEEE 802.15.4 packets with a maximum payload of 127 bytes. Moreover, 6LoWPAN requires a low-power border router at the edge to translate incoming and outgoing packets to either IPv6 or 6LoWPAN network.

RPL

RPL is an intra-domain routing protocol for low-power wireless mesh networks [35]. RPL works by constructing the network into a Directed Acyclic Graph (DAG), rooted at the gateway. Root, or sink, builds the Destination Oriented Directed Acyclic Graph (DODAG) shown in Figure 2.8. RPL is a distance vector protocol where nodes regularly advertise their distance to the DODAG. This allows neighbors to compute their own distance. Motes broadcast Destination Advertisement Object (DAO) messages to allow neighbours to find their Default Parent (DP). DP is their route towards the DODAG. DODAG builds a network, where all motes have a rank. A high rank indicates a large distance and low rank imply shorter distance to DODAG.

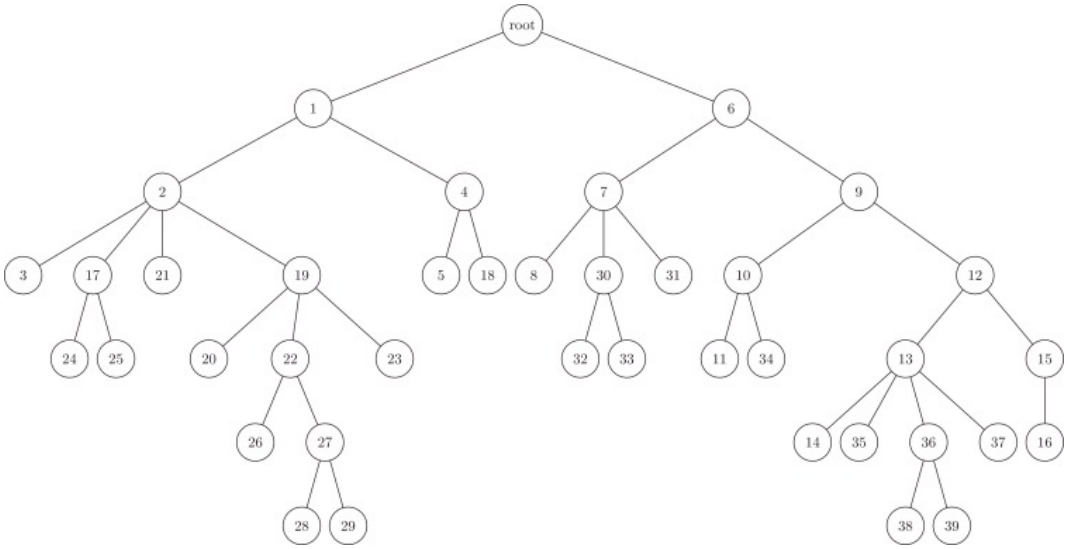


Figure 2.8: Example of a RPL network build from root retrieved from [7]

Contrained application protocol

Constrained Application Protocol (CoAP) turns low-power wireless devices into a web server and a browser allowing web-like interactions [36]. Hence, a mote utilizing CoAP can publish its sensor readings onto servers on the Internet. Furthermore, CoAP has a large open-source library allowing it to support different applications. CoAP layer is not utilized in this thesis.

2.5.2 Tracks

What is referred to as a DetNet flow is introduced in 6TiSCH architecture as Complex Tracks [16]. Which in principle, are reserved resources from source to sink.

In essence, cells are reserved for specific traffic, and no other form of traffic can access these cells. Multiple cells bundled together increase probability of successful transmission within a slotframe. Complex Tracks are shaped as a DAG, and support multi-path forwarding and route around failures [16]. An illustration of a track is provided in Figure 2.9.

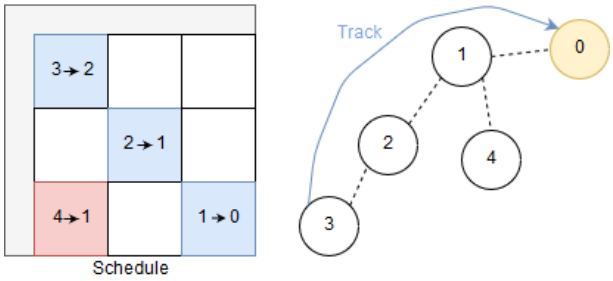


Figure 2.9: Illustration of a 6TiSCH track from 3 to 0

Complex Tracks are utilized to ensure application data have resources available from source to sink. In Complex Tracks, only allowed traffic are scheduled. This is necessary to ensure no randomness in data traffic, enabling determinism.

2.6 Packet error rate

As mentioned in previous Section 2.3 and Section 2.5, high reliability is desired by DetNet and 6TiSCH. What is more, high reliability depend on Packet Error Rate (PER). This section present relationship between Bit Error Rate (BER) and PER. BER is often expressed in percent, and can be considered an approximate of the bit error probability. PER depend on BER and number of bits (e.g the packet length). PER can be calculated by:

$$PER = 1 - (1 - P_b)^{n_{bits}} \tag{2.3}$$

Where

P_b	Probability of bit error
n_{bits}	Packet size in bits

Packet delivery ratio

Flipping PER and describing it as a Packet Delivery Ratio is done in Equation 2.4.

$$PDR = 1 - PER \tag{2.4}$$

Moreover, in a wireless setting it is common make an abstraction of the physical layer when working with higher layers and give a static PDR. But, working with different packet sizes, static PDR do not represent the same BER. Thus, it gives an unrealistic representation of a smaller packet compared to a larger packet. A workaround is shown in Equations 2.5 - 2.9. PER_l denotes a large packet and PER_s a small packet. Correspondingly, packet lengths are denoted as L_l and L_s.

$$PER_l = 1 - (1 - P_b)^{L_l} \tag{2.5}$$

This can be written as:

$$1 - P_b = (1 - PER_l)^{\frac{1}{L_l}} \tag{2.6}$$

Similarly to PER_l, packet error rate for a smaller packet is:

$$PER_s = 1 - (1 - P_b)^{L_s} \tag{2.7}$$

Substituting Equation 2.6 into Equation 2.7 yields:

$$PER_s = 1 - (1 - PER_l)^{\frac{L_s}{L_l}} \tag{2.8}$$

Next, utilizing relationship seen in Equation 2.4:

$$PDR_s = 1 - PDR_l^{\frac{L_s}{L_l}} \tag{2.9}$$

Putting specific packet sizes show a 127 bytes packet with a PDR of 70% equals to a PDR of 93.7% for a 23 bytes packet. This method is applied to PER values of 30%, 20% and 10%, with the results listed in Table 2.2. As PER is static in this example, BER can be derived from Equation 2.3.

PER	BER	PDR_s	PDR_l
90%	0.0003483	93.7%	70%
20%	0.0002178	96.1%	80%
10%	0.0001028	98.1%	90%

Table 2.2: PDR values for PDR_s and PDR_l

Figure 2.10 shows relationship between PDR_s 23 bytes and PDR_l 127 bytes. The reason these specific packet lengths are used is discussed further on.

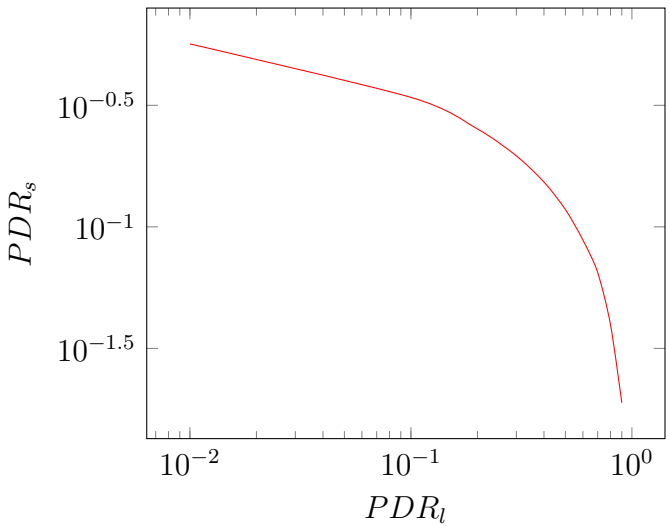


Figure 2.10: Relationship between PDR_s (23 bytes) and PDR_l (127 bytes) in loglog.

2.7 Latency

As presented in Subsection 2.3.1, a DetNet primary goal is achieving minimum and maximum end-to-end latency. In a TSCH network maximum and minimum latencies can be calculated. This is due to the fact that the value of a slot and a slotframe never changes while a network is operational. Thus, maximum latency L_{max} and minimum latency L_{min} can be calculated with:

$$L_{max} = S_f \cdot T_s \cdot m \cdot H \tag{2.10}$$

And,

$$L_{min} = T_s \cdot H \tag{2.11}$$

Where,

T_s	Length of timeslot in seconds
S_f	Slotframe length (number of slots)
m	Number of allowed retransmissions
H	Number of hops

In L_{max} and L_{min} assume next hop node has a cell in the following timeslot. Thereby, in L_{min} the S_f and m are not needed as the packet is successful and do not need a retransmission in the next slotframe. As mentioned in Section 1.3, one of the objectives of this thesis is introducing a delay τ in the replication mechanism.

This will be discussed in details in Chapter 3, but for now maximum delay, L_{max} with replication delay τ is calculated with:

$$L_{max} = (S_f \cdot T_s \cdot m \cdot H) + \tau \tag{2.12}$$

Where,

τ	Sending delay of replicated copy in slots

2.8 Analytic approach

This section is based on work done in [37]. In [37] the author derive a method to calculate expected number of transmissions over multiple hops. Furthermore, when sending packets, error occurs, and to analyze packet loss analytically a network of infinite nodes and node density must be considered. Routing in a network this size is virtually impossible, but for an analytic approach assume all nodes have a mechanism to relay a packet to the furthest node in a direction limited by its range. Thus, hops required to send a packet to a sink located at distance D from the sender is:

$$H = \left\lceil \frac{D}{d} \right\rceil \tag{2.13}$$

Where,

$\lceil x \rceil$	Denotes the ceiling function
d	Range of node
D	Total distance from sender to sink

Given this example, denote sender as A. A has a line topology depicted in Figure 2.11 to sink. Next, each hop i, where $i \in \{1...(H-1)\}$ has same length d. Moreover, assume path loss only depend on distance and that nodes have equal transmitting power. H^{th} hop depend on distance from A to sink, and node range. For the sake of this example assume H hops having identical length d. Finally, assume only one node transmits at a time causing zero interference between nodes, and that there are no external sources of interference.

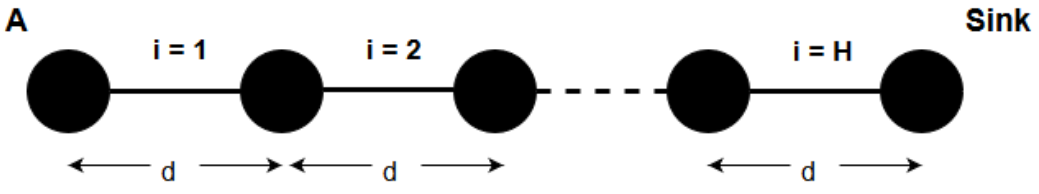


Figure 2.11: Line topology from A to Sink

2.8.1 Expected number of transmissions

Assume premise introduced in the previous section. X_i is number of transmissions on the i^{th} hop for each packet sent from node A towards sink. Realistically, BER is not static and varies due to a range of factors, but follow the same 'ideal' concept and assume BER for each hop is P_b:

$$P_{bi} = P_b \forall i \in \{1...H\} \tag{2.14}$$

Assume probability of individual bit errors are independent from each other. Next, as TSCH utilizes ACKs a transmission is successful upon reception of an ACK. Transmission is deemed unsuccessful if the node do not receive an ACK, which triggers a retransmission. Number of retransmission attempts are denoted as m and if the number of retransmissions passes m, the packet is discarded. Two scenarios can cause a retransmission when utilizing ACK:

- Packet was lost during transmission
- Packet was successful, but ACK transmission failed

Thereby, for a packet transmission to be confirmed as successful, all bits in the ACK and the data packet has to be transferred without errors as shown in Section 2.6. Taking two-way transmissions into account let q be packet failure rate, probability that a transmission fails.

$$q = 1 - (1 - P_b)^{L_{Packet}+L_{Ack}} \tag{2.15}$$

Where,

L_{Packet}	Length of packet in bits
L_{Ack}	Length of ACK in bits

Expected number of transmissions at first hop

Thus, X_{first}, expected number of transmissions at first hop is determined by probability of a successfully transmission for each attempt m. This can be expressed as:

$$E\{X_{first}\} = 1{\cdot}q^0(1-q)+2{\cdot}q^1(1-q)+3{\cdot}q^2(1-q)+...+m{\cdot}q^{m-1}{\cdot}(1-q)+m{\cdot}q^m \tag{2.16}$$

Where,

m	Maximum number of transmissions
q	Probability of an unsuccessful transmission
$(1-q)$	Probability of a successful transmission

or, it can be expressed as:

$$E\{X_{first}\} = \sum_{i=1}^{m} \Big(i \cdot q^{i-1} \cdot (1-q) \Big) + m \cdot q^m \tag{2.17}$$

Where, as earlier, m is maximum number of transmission attempts. Thus, $(m-1)$ is maximum number of retransmissions. Seeing as in Equation 2.17, $(1-q)$ is a constant it can be factored out. By denoting $S = \sum_{i=1}^{m} i \cdot q^{i-1}$ Equation 2.17 can be written as:

$$E\{X_{first}\} = S \cdot (1-q) + m \cdot q^m \tag{2.18}$$

The sum, S, can thereby be expressed as:

$$S = 1 \cdot q^0 + 2 \cdot q^1 + ... + m \cdot q^{m-1} \tag{2.19}$$

And by subtracting $S \cdot q$ from S:

$$S(1-q) = 1 \cdot q^0 + 2 \cdot q^1 + ... + m \cdot q^{m-1} \tag{2.20}$$

$$-(1 \cdot q^1 + 2 \cdot q^2 + ... + m \cdot q^m \tag{2.21}$$

$$= (q^0 + q^1 + q^2 + ... + q^{m-1}) - m \cdot q^m \tag{2.22}$$

Equation 2.22 is a geometric series, and can be written as:

$$\frac{1 - q^{(m-1)+1}}{1-q} = \frac{1-q^m}{1-q} \tag{2.23}$$

S can be simplified to:

$$S = \frac{1-q^m}{(1-q)^2} - \frac{m \cdot q^m}{(1-q)} \tag{2.24}$$

Returning S into Equation 2.17:

$$E\{X_{first}\} = \left(\frac{1-q^m}{(1-q)^2} - \frac{m \cdot q^m}{(1-q)} \right)(1-q) + m \cdot q^m = \frac{1-q^m}{1-q} \tag{2.25}$$

Equation 2.25 represent expected number of transmissions:

$$E\{X_{first}\} = \frac{1-q^m}{1-q} \tag{2.26}$$

Limits of Equation 2.26 can be tested for the extremes when $q = 0$ and $q = 1$ which means either all packets or no packets are successful.

$$\lim_{q \to 0} \frac{1-q^m}{1-q} = 1 \tag{2.27}$$

and by utilizing L'Hopital's rule for limit $q \to 1$,

$$\lim_{q \to 1} \frac{1 - q^m}{1 - q} = \lim_{q \to 1} \frac{m \cdot q^{m-1}}{1} = m \tag{2.28}$$

This show when q is 0 no packets are expected to fail and first transmission attempt is successful. On the other hand, when q is 1 all m transmissions are expected to fail. Equation 2.27 and Equation 2.28 show Equation 2.26 hold at its limits.

Multiple hops

In contrast to Equation 2.26, most transmission scenarios for a packet involves several hops. End-to-end probability at the i^{th} hop depends on the probability that the packet has traversed $(i-1)$ previous hops, $P(S_{i-1}) = (1 - q^m)^{i-1}$, where $P(S_i)$ is probability of a successful transmission at hop i. At the i^{th} hop number of expected transmissions are:

$$E\{X_i\} = E\{X_{first}\} \cdot P(S_{i-1}) = E\{X_{first}\} \cdot (1 - q^m)^{i-1} \tag{2.29}$$

Given by Equation 2.29, expected number transmissions over H hops are:

$$\sum_{i=1}^{H} E\{X_i\} = E\{X_{first}\} \cdot \sum_{i=1}^{H} (1 - q^m)^{i-1} \tag{2.30}$$

Seeing as $\sum_{i=1}^{H} (1 - q^m)^{i-1}$ is a geometric series, and by denoting $(1 - q^m)$ as k:

$$k^0 + k^1 + k^2 + ... + k^{i-1} \tag{2.31}$$

Equation 2.31 can be written as:

$$\sum_{i=1}^{H} k^{i-1} = \sum_{i=0}^{H-1} k^i = \frac{1 - k^H}{1 - k} \tag{2.32}$$

Returning $(1 - q^m)$ into Equation 2.32

$$\frac{1 - (1 - q^m)^H}{1 - 1 + q^m} = \frac{1 - (1 - q^m)^H}{q^m} \tag{2.33}$$

Thus, Equation 2.30 is simplified to:

$$E\{X_i\} = E\{X_{first}\} \cdot \frac{1 - (1 - q^m)^H}{q^m} \tag{2.34}$$

At limit of Equation 2.30, $q \to 1$ total number of transmissions are expected to be m as it never get past first hop. Conversely, when $q \to 0$ only one transmission

is needed per hop and expected transmissions are equal to H to reach sink. Limits of Equation 2.30 are found by multiplication law of limits (if limits exist):

$$\lim_{q\to\ c}\left(E\left\{X_i\right\}\cdot\frac{1-(1-q^m)^H}{q^m}\right)=\lim_{q\to\ c}E\left\{X_i\right\}\cdot\lim_{q\to\ c}\frac{1-(1-q^m)^H}{q^m} \tag{2.35}$$

Where,

$$c \quad \text{is a constant}$$

As $\lim_{q\to\ c}$ is know for $c=0$ and $c=1$, only Equation 2.33 need to be analyzed. Thus, by power law of limits (if it exists):

$$\lim_{q\to\ 1}\frac{1-(1-q^m)^H}{q^m}=1 \tag{2.36}$$

and by utilizing L'Hopital's rule for limit $q\to 0$,

$$\lim_{q\to\ 0}\frac{1-(1-q^m)^H}{q^m}=\lim_{q\to\ 0}\frac{Hmq^{m-1}(1-q^m)^{H-1}}{mq^{m-1}} \tag{2.37}$$

Next, this can be shortened:

$$\frac{Hmq^{m-1}(1-q^m)^{H-1}}{mq^{m-1}}=H(1-q^m)^{H-1} \tag{2.38}$$

And now, by inserting q = 0:

$$H(1-q^m)^{H-1}=H \tag{2.39}$$

Summarizing, limits at Equation 2.30 are:

$$\lim_{q\to\ 1}\sum_{i=1}^{H}E\{X_i\}=m\cdot 1=m \tag{2.40}$$

And:

$$\lim_{q\to\ 1}\sum_{i=1}^{H}E\{X_i\}=1\cdot H=H \tag{2.41}$$

This shows if q = 1, no error occur and number of transmissions are equal to number of hops. Next, if q = 0 number of transmissions are equal to number of allowed transmission attempts. This indicates that transmission has failed. To summarize, expected number of transmissions from a source to a sink depend on number of transmissions attempts, hops and PER.

2.9 Radio transceivers

IEEE 802.15.4 devices require a transmitter and a receiver to communicate. A transmitter takes digital information and translates it into radio waves propagating the wireless medium, while the receiver interpreters waves and reproduce emitted information. Task of sending and receiving are often grouped into a single device called *transceiver*. These transceivers operate similarly to micro-controllers as they have different operating modes, such as transmit, receive, and idle mode. To reduce energy consumption of each node, transceivers should be turned off most of the time. Moreover, transceivers should be activated only when needed to achieve a low duty cycle.

2.9.1 Energy consumption

Power consumption in a given output configuration can be expressed as

$$P_i = U \cdot I_i \tag{2.42}$$

Where,

P_i	Watts
U	Supply voltage in volts
I_i	Current consumption of node

Thus, energy consumption can be expressed as

$$E_{tx} = P_i \cdot T_{tx} \tag{2.43}$$

Where,

E_{tx}	Joules
T_{tx}	Output power of transmitter/receiver

However, these equations do not take all factors into consideration and does a generalization. In principle, energy consumed by a transceiver is from two sources. Firstly due to RF signal generation, which in return depend mostly on chosen modulation, target distance and transmission power, P_T, radiating from the antenna. Secondly, due to electronic components used for frequency synthesis, frequency conversion, filters and so on [38]. However, factors composing the second source are basically constants. Thus, most crucial parameter is choice of P_T. When P_T has been set power consumption is affected by time spent transmitting. Energy to send a packet that is n-bits long (including headers) depend on time it takes to

transmit, set by bit rate, coding rate and total power consumed during transmission. In addition, transceivers has to be turned on before transmission if they are in sleep mode, adding some start-up cost. Energy costs can be calculated by [38]:

$$E_{tx}(n, R_{code}, P_{amp}) = T_{start}P_{start} + \frac{n}{RR_{code}}(P_{txElec} + P_{amp}) \tag{2.44}$$

Where,

E_{tx}	Total energy spent transmitting
R_{code}	Coding rate
P_{amp}	Power of transmitting amplifier
T_{start}	Start-up time
P_{start}	Start-up power
R	Bit rate
P_{txElec}	Power usage of other circuitry
n	Packet length

Moreover, receiver can, as transmitter, be turned on or off. When a receiver is turned on it can be idle, or be actively receiving a packet. When idle it observe the channel and are ready to receive incoming packets. Power usage between receiving and idle mode is negligible and can be assumed to be zero. Energy consumption when receiving a packet can be calculated by [38]:

$$E_{rx}(n, R_{code}, P_{amp}) = T_{start}P_{start} + \frac{n}{RR_{code}}P_{rxElec} + nE_{decBit} \tag{2.45}$$

Where,

E_{rx}	Total energy spent receiving
R_{code}	Coding rate
P_{amp}	Power of transmitting amplifier
T_{start}	Start-up time
P_{start}	Start-up power
R	Bit rate
P_{rxElec}	Power usage of other circuitry
n	Packet length
nE_{decbit}	Energy spent decoding bits

2.9.2 Total time spent transmitting and receiving

Energy consumption calculation method presented in Subsection 2.9.1 is affected by standards and hardware designs, and for the purpose of this thesis not under

analysis. However, a parameter that can be affected is the time the radio is in transmitting and receiving mode. Total time spent transmitting T_{tx} depend on number packets sent and time it takes to send each packet. What is more, time to send each packet depend on packet size. Thereby, it is possible to derive a method to calculate T_{tx} and T_{rx} only considering ACK and data packets.

Transmitting

$$T_{tx} = N_{txPacket} \cdot \frac{L_{Packet}}{R} + N_{rxACK} \cdot \frac{L_{ACK}}{R} \tag{2.46}$$

Where,

$N_{txPacket}$	Number of transmitted packets
N_{rxACK}	Number of ACKs received
L_{Packet}	Length of packet in bits
L_{ACK}	Length of ACK in bits
R	Data rate in $bits/s$

Receiving

Total time spent receiving T_{rx} depend on number of incoming packets and time it takes to receive each packet. Thereby, it is possible to calculate T_{rx} only considering ACKs and data packets with

$$T_{rx} = N_{rxPacket} \cdot \frac{L_{Packet}}{R} + N_{txACK} \cdot \frac{L_{ACK}}{R} \tag{2.47}$$

Where,

$N_{rxPacket}$	Number of received packets
N_{txACK}	Number of ACKs transmitted
$L_{Packets}$	Length of packet in bits
L_{ACK}	Length of ACK in bits
R	Data rate in $bits/s$

2.9.3 Radio up time

T_{tx} and T_{rx} is equal in terms of time, an abstraction can be made calling total radio up time for RT_{ux}.

$$RT_{ux} = 2 \cdot \left(N_{Packets} \cdot \frac{L_{Packet}}{R} + N_{Ack} \cdot \frac{L_{Ack}}{R} \right) \tag{2.48}$$

RT_{ux} returns total time transceiver is active.

2.9.4 Expected radio up time

In Subsection 2.9.2 PER is not taken into consideration and it assume a single transmission when calculating time spent transmitting. However, this is not realistic and expected radio up time ER_{ux} depend on expected number of transmissions.

Single Hop

ER_{ux} for a single hop is given by:

$$ER_{ux} = 2 \cdot \left(N_{Packet} \cdot \frac{L_{Packet}}{R} + N_{ACK} \cdot \frac{L_{ACK}}{R} \right) \cdot \left(\frac{1 - q^m}{1 - q} \right) \tag{2.49}$$

or simply put:

$$ER_{ux} = RT_{ux} \cdot E\{X_{first}\} \tag{2.50}$$

Where,

RT_{ux}	Radio up time in seconds
$E\{X_{first}\}$	Expected number of transmissions at first hop derived in Subsection 2.8.1

2.9.5 Energy consumption in TSCH networks

A method of calculating lifetime of a TSCH node is derived in [2]. A TSCH node has six different type of timeslots as presented in Table 2.3.

State	**Description**
Idle	Node idle listens. This is a RX state where nothing is received. Hence it only listens for duration of guard time.
Sleep	Node Deep Sleeps. Slot is off, no CPU nor radio activity due to communication.
TxDataRxAck	Node transmits a packet and receives an ACK for it.
TxData	Node sends a broadcast packet not requiring ACK.
RxDataTxAck	Node receives a packet and responds with an ACK.
RxData	Node receive a frame not requiring ACK.

Table 2.3: The six different timeslots in a IEEE 802.15.4 network retrived from [2]

By utilizing the energy model of each slot, it is possible to determine energy consumption and compute expected lifetime of the node battery. Next, to calculate their energy consumptions each slots contribution needs to be considered. For simplicity and minimizing text, all slots have new denotation as shown Table 2.4.

What is more, in Equations subscripts a point to available slots, and subscript u expresses utilized slots. N indicates *Number of*. In addition, all charges Q are given in μCoulombs (C).

State	**Denotation**
Sleep	N_{sleep}
TxDataRxAck	N_{TxRx}
TxData	N_{Tx}
RxDataTxAck	N_{RxTx}
RxData	N_{Rx}

Table 2.4: Denotation change for timeslots

$$Q_{FIdle} = \frac{\Big((N_{aRxTx} - N_{uRxTx}) + (N_{aRx} - N_{uRx})\Big) \cdot Q_{idle}}{PDR} \tag{2.51}$$

Q_{FIdle} gives contribution of idle slots to total charge drawn at a slotframe. Idle slots is when RxData or RxDataTxAck are active but nothing is received.

$$Q_{FSleep} = \Big(N_{sleep} + (N_{aTxRx} - N_{uTxRx}) + (N_{aTx} - N_{uTx})\Big) \cdot Q_{Sleep} \tag{2.52}$$

Then, Q_{FSleep} defines contribution of sleep slots.

$$Q_{FTxRx} = \frac{N_{uTxRx} \cdot \left(\frac{NBSent}{MaxPktSz} \cdot Q_{Tx} + (Q_{TxRx} - Q_{Tx})\right)}{PDR} \tag{2.53}$$

Where,

$NBSent$	Number of bytes sent
$MaxPktSz$	Maximum packet size

Q_{FTxRx} defines contribution of sending a packet and receiving an ACK. Specifically, effect packet size has on energy consumption.

$$Q_{FTx} = N_{uTx} \cdot \left(\frac{NBSent}{MaxPktSz} \cdot Q_{Tx}\right) \cdot PDR \tag{2.54}$$

Moreover, Q_{FTx} describes energy consumption of slots that transmits, but do not require an ACK.

$$Q_{FRxTx} = N_{uRxTx} \cdot \left(\frac{NBSent}{MaxPktSz} \cdot Q_{Rx} + (Q_{RxTx} - Q_{Rx})\right) \tag{2.55}$$

Q_{FRxTx} computes contribution of receiving a packet and transmitting an ACK. PDR and number of bytes are taken into consideration.

$$Q_{FRx} = N_{uRx} \cdot \frac{NBSent}{MaxPktSz} \cdot Q_{Rx} \tag{2.56}$$

Q_{FRx} describes contribution of slots that do not need to transmit an ACK.

$$Q_{slotframe} = QFIdle + QFsleep + Q_{FTxRx} + Q_{FTx} + Q_{FRxTx} + Q_{FRx} \tag{2.57}$$

Then, $Q_{slotframe}$ sums total charge drawn during a slotframe.

$$lf = \frac{B_{capacity} \cdot 3.6}{Q_{slotframe}} \cdot \frac{Length_{slot} \cdot Length_{slotframe}}{3600 \cdot 24} \tag{2.58}$$

At last, lf computes battery lifetime of a node in days, assuming a 3.6V power supply. $B_{capacity}$ is battery capacity given in μAh.

2.10 Funneling effect

When working with WSNs, there is a higher load on motes closer to sink as compared to motes further away [8]. This effect is visualized in Figure 2.12. Data packets are traversing towards sink and hence increasing load on motes closer to it. Thus, motes further away have less traffic, and less funneling effect.

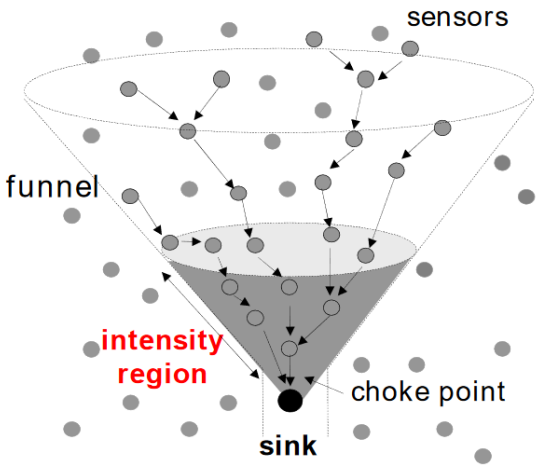


Figure 2.12: Illustration funneling effect on nodes closer to sink retrieved from [8]

Chapter 3

Proposal

This thesis proposes a novel Reverse Packet Elimination (RPE) algorithm. RPE was developed with DetNet architecture presented in Section 2.3 in mind and utilizes the proposed elements explicit routes, service protection, in-order delivery, packet replication and packet elimination.

3.1 Reverse Packet Elimination

In short, with RPE source replicates a packet and send copies towards sink, along two different paths. But, when a copy arrives at the sink it triggers a RPE packet down the opposite path from which it came. Purpose of RPE is to search for the other copy on its way upstream and eliminate it. Subsequently, the second copy can be withheld with an adjustable delay τ shown in Figure 3.1.

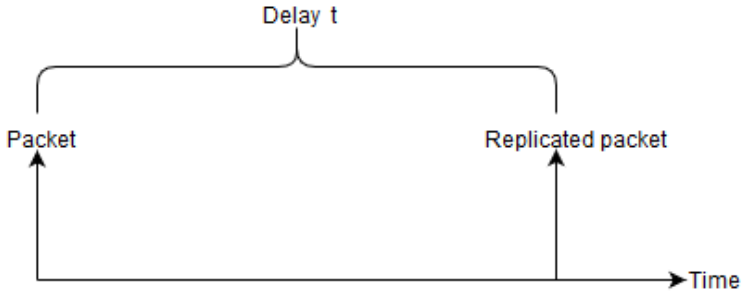


Figure 3.1: Illustration of how the replicated is withheld by τ

3.1.1 Advantages of Reverse Packet Elimination

As shown in Section 2.6 packet size matter in terms of PDR. Thus, utilizing a packet significantly smaller downstream from sink to source have a higher delivery

ratio than a larger upstream packet. Next, as derived in Subsection 2.9.2 smaller packets require less radio up time. Finally, as PDR is higher and radio up time is lower, energy consumption can be decreased as shown in Subsection 2.9.5. This proposal is described in six steps:

Step 1: A Path Computation Element (PCE) schedules 6TiSCH tracks in daisy chains with two paths, Path A and Path B. Path A is by design scheduled to send first. As shown in Figure 3.2, blue cells make up a track from source (7) to sink (0).

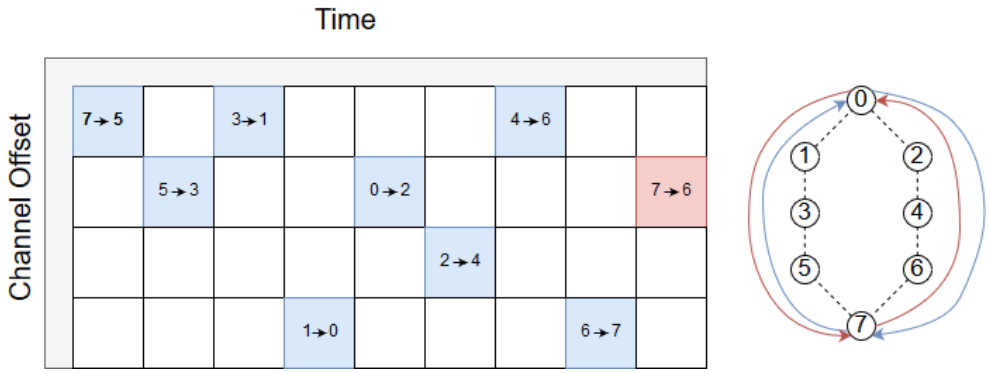


Figure 3.2: Illustration of PCE scheduling tracks from source to sink

Step 2: Source replicates a packet and sends copies up two disjoint paths, Path A and Path B, which both has explicit routes towards sink as shown in Figure 3.3. The copy along Path B however, is withheld by a delay τ dependent on upper bound latency of an application. For instance, the control variable utilized in the feedback loop in Section 1.2. In short, it can be minimum 1 timeslot with no limit on maximum replication delay.

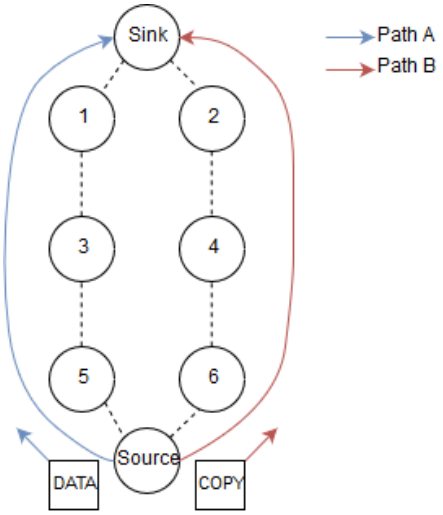


Figure 3.3: Source creates two copies and send one each disjoint path

Step 3: Copies propagates disjoint paths towards the sink. If both Path A and Path B arrives at sink before sink has chance to send a RPE packet, the packet arriving last is discarded. This step is illustrated in Figure 3.4.

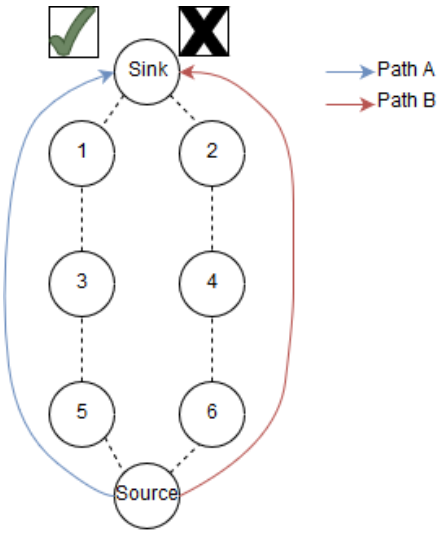


Figure 3.4: Packets are received at sink before RPE packet is sent

Step 4: If they do not arrive at the same time, sink sends a RPE packet down the path the copy did not arrive from as shown in Figure 3.5. If Path A arrives first the RPE packet is sent down Path B and vice versa.

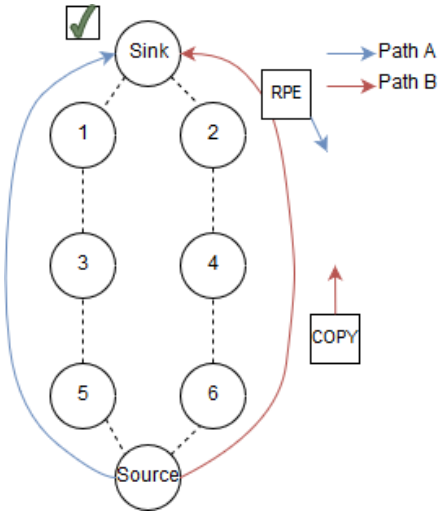


Figure 3.5: Sink send a RPE packet down Path B

Step 5: A RPE packet then traverse the network downwards toward the source searching TX queues for a packet going upstream. When it locates a packet with identical sequence number it tells the mote to drop it as has been delivered to the sink. This is depicted in Figure 3.6.

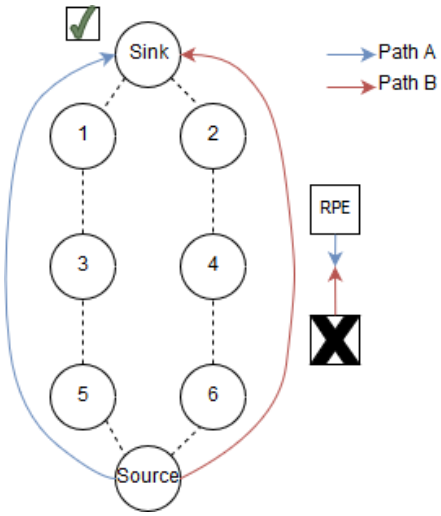


Figure 3.6: RPE packet locates a upstream packet and drops it

Step 6: If a RPE packet manages to arrive at source (looping back) before the other copy is sent, source do not send the second copy as shown in Figure 3.7.

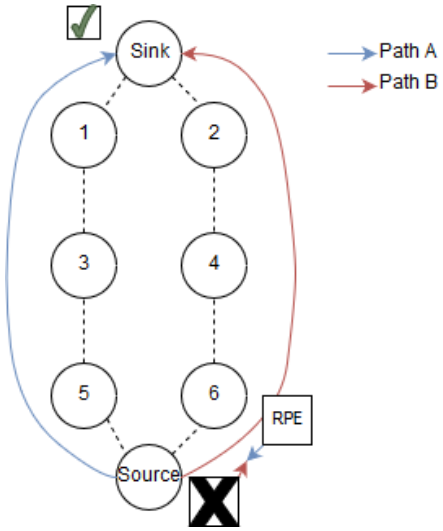


Figure 3.7: A RPE packet sent from sink down Path B makes it back to source eliminating the copy before it is sent

3.1.2 Reverse Packet Elimination frame format

RPE frame format follow guidelines of IEEE 802.15.4 standard [9]. A RPE amendment would require use of Information Element (IE) header. An IE header is a optional field in IEEE 802.15.4 general frame format.

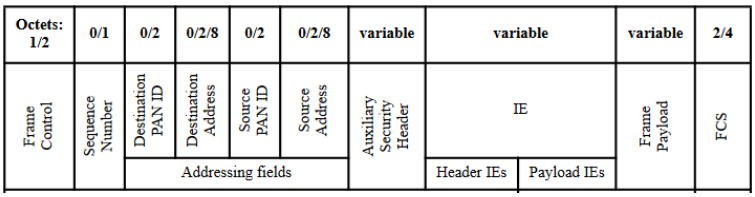


Figure 3.8: MAC frame format from IEEE 802.15.4 standard retrieved from [9]

Moreover, a RPE frame would follow general MAC frame format as presented in Figure 3.8. The only requirement is usage of IE header shown in Table 3.1.

Bits: 0-6	**7-14**	**15**	**Octets: 0-127**
Length	Element ID	Type = 0	Content

Table 3.1: Information Element header format from IEEE 802.15.4 standard

In the header, Element ID can be utilized to indicate it is a RPE packet. Then, content contains sequence number RPE is looking for. Frame format of a RPE

packet would require fields and sizes as represented in Table 3.2. This sums up to a total of 23 bytes. A RPE frame do not require use of an IE payload has all needs are satisfied by an IE header.

Octets: 1	**1**	**8**	**8**	**3**	**2**
Frame control	Sequence number	Destination address	Source address	IE	FCS

Table 3.2: RPE frame format

Effectively this means a RPE frame is similar to a standard frame, but the payload of a RPE packet is an IE header. For this reason, RPE is payload 23 bytes, while a standard frame has a variable payload with a maximum of 127 bytes. Moreover, RPE ACKs have same format as a IEEE 802.15.4 ACK depicted in Figure 3.3

Octets: 2	**1**	**2/4**
Frame Control	Sequence Number	FCS

Table 3.3: Acknowledgement frame format

3.1.3 Selection of delay τ

This section present selected delays τ. Moreover, an explanation as to why these specific τ were preselected are provided. Firstly, selected delays are presented in Table 3.4. Furthermore, as mentioned earlier τ is always given in slots, and it indicates the delay from when the first copy was sent. As mentioned in Section 2.2.2 a slotframe is commonly 101 slots. Hence, if $\tau > 101$ it indicates the copy is scheduled at least one slotframe or more later.

τ
1
8
816
1624

Table 3.4: List of selected τ

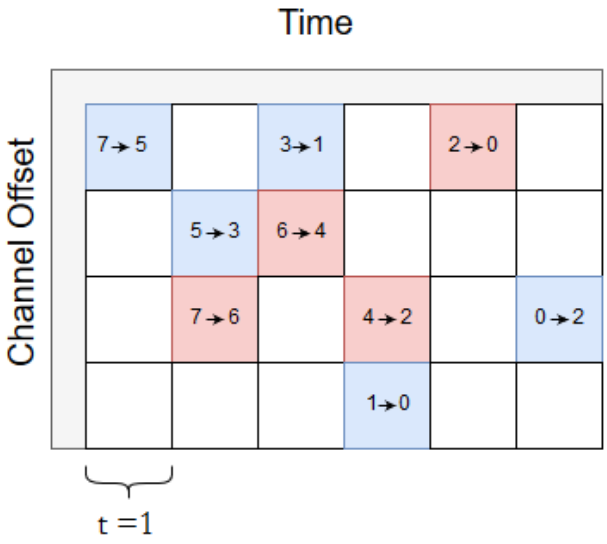


Figure 3.9: Scheduling a delay of $\tau = 1$

Next, evaluating selection of delays. Starting with $\tau = 1$, which trigger source to always send two copies, and presumably have lowest latency due to lowest replication delay. Moreover, with such a low delay, a packet always traverse Path A and Path B simultaneously. With $\tau = 1$, sink can not send a RPE packet down when Path A arrive as it needs wait one slot to ensure no collision with Path B. This is shown in Figure 3.9 where blue track wait for red to arrive at sink (0).

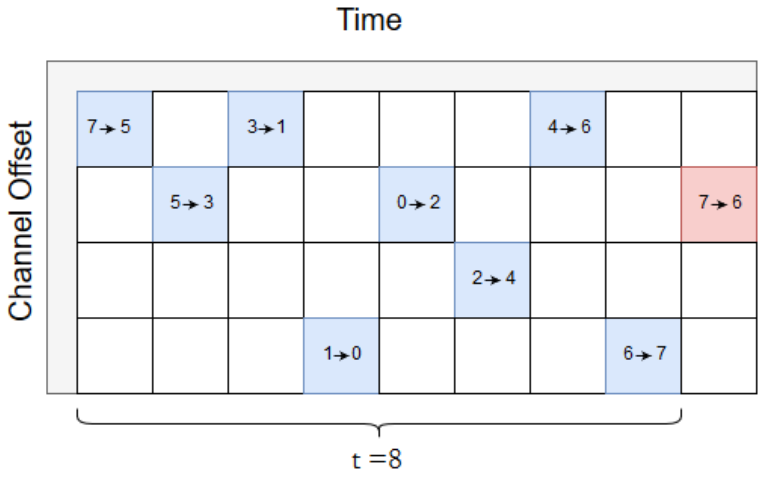


Figure 3.10: Scheduling a delay of $\tau = 8$

Next, $\tau = 8$ has the trait that if no retransmissions occur at Path A, or in the reverse path back to the source, Path B packet will not be sent. This is illustrated

in Figure 3.10 which show schedule for the network in Figure 3.6. With this in mind, in an ideal world with $\tau = 8$, Path A packet has enough time to loop back and stop Path B packet. Then, looking at $\tau = 1624$ which is the theoretical maximum latency of the packet looping back to the source. This is a wait time of 16 slotframes. In essence, Path A packet is given the chance to utilize all transmission attempts before Path B packet is transmitted. As seen in Subsection 2.8.1, a transmission fail if a packet was lost during transmission or if the ACK fails. Hence, motes have several attempt to try ensure it is successfully transmitted. However, if a transmission fails in the current configuration, a mote has to wait until the next slotframe iteration before trying again. Transmission attempts are set to 4 as recommended in [39]. There is an upper limit to retransmission attempts to ensure packets are dropped if they never succeed. Finally, $\tau = 824$ was selected to see if there is a difference from the long delay $\tau = 1624$. With $\tau = 824$ delay is cut from 16 slotframes to 8 slotframes.

Topology specific

With delays from Table 3.4, all except $\tau = 1$ are topology specific. Meaning, $\tau = 8$ depend on number of hops. In short, $\tau = 8$ represent 4 hops to sink and 4 hops in return to source. Adding one hop would increase delay to $\tau = 10$. What is more, theoretical maximum latency depend on number of hops meaning $\tau = 1624$ is topology specific as well.

3.1.4 Within a slotframe

At first, the idea of having a delay $1 < \tau < 101$ was presumed the ideal solution in terms of delay within a slotframe. But after a closer look at how TSCH slotframes behave, ideal delay is $1 < \tau < 8$. As everything after 8 slots is extra unnecessary delay due to the number of hops. This is due to the fact, as mentioned earlier, if a transmission fails, the retransmission has to wait 1 slotframe iteration. Meaning, if a transmission fail at slot 10 and then having a copy wait until slot 17, is 7 slots of unnecessary delay as presented in Figure 3.11.

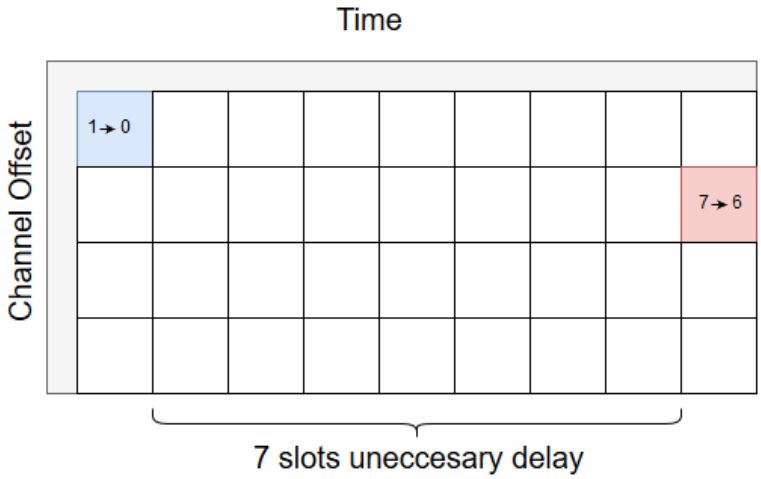


Figure 3.11: Unnecessary waiting delay within a slotframe

Chapter 4

Method

This chapter present methods of how to investigate the Reverse Packet Elimination algorithm. In fact, several approaches can be utilize to analyze RPE. Most common methods to study networks involve analytic approaches, simulations or testbeds. However, as 6TiSCH stack is still under development and not a finished standard, there are limited options for testbeds. For that reason, an analytic and simulation approach are selected.

4.1 Simulator selection

When simulating networks, there are a few well-known options which tends to be utilized. Commonly, *OMNet++* [40] and *ns-3* [41] are used for simulating network behavior, but as of late 2018 none of them have implemented 6TiSCH. Although, *ns-3* mention TSCH is to be implemented. Next, *Contiki-NG* with the Cooja emulator has 6TiSCH implemented and is a viable option [42]. In addition, Cooja emulator is written in C, a low level programming language making it easier to port to real motes. However, chosen simulator is the 6TiSCH simulator [43] due to more personal experience with Python, as opposed to C, and the fact that another student uses the same tool allowing for discussions. Moreover, the simulator was created by some of the members of the 6TiSCH working group, and is designed to minimize typical simulation drawbacks by careful abstractions specific to 6TiSCH. My findings are supported by [3], and are presented in Table 4.1.

Simulator	Learning curve	Scalability	6TiSCH implementation	Standard-compliant
ns-3	High	Medium	None	N/A
OMNet++	High	Medium	None	N/A
TOSSIM	Medium	High	None	N/A
Cooja (Emulator)	High	Low	Yes (partial)	Partially
OpenSim (Emulator)	High	Low	Yes	Yes (Byte-accurate)
6TiSCH Simulator	Low	High	Yes	Yes (Behavioral)

Table 4.1: General overview of options in regards to 6TiSCH derived from [3]

4.2 6TiSCH Simulator

The following information is derived from [3]. The 6TiSCH Simulator is a discrete-event simulator written in Python encompassing 6 core files. Main component is *Mote* where the 6TiSCH stack is implemented. *Mote* is configured by *SimSettings* which holds input parameters a user can tweak. Next, *Mote* generates metrics for *SimStats* and program events scheduled and processed by *SimEngine*. Example of such events are TXs and RXs influenced by *Propagation*, which depend on *Topology*. Moreover, internal architecture is illustrated in Figure 4.1. The following subsections describe the core files.

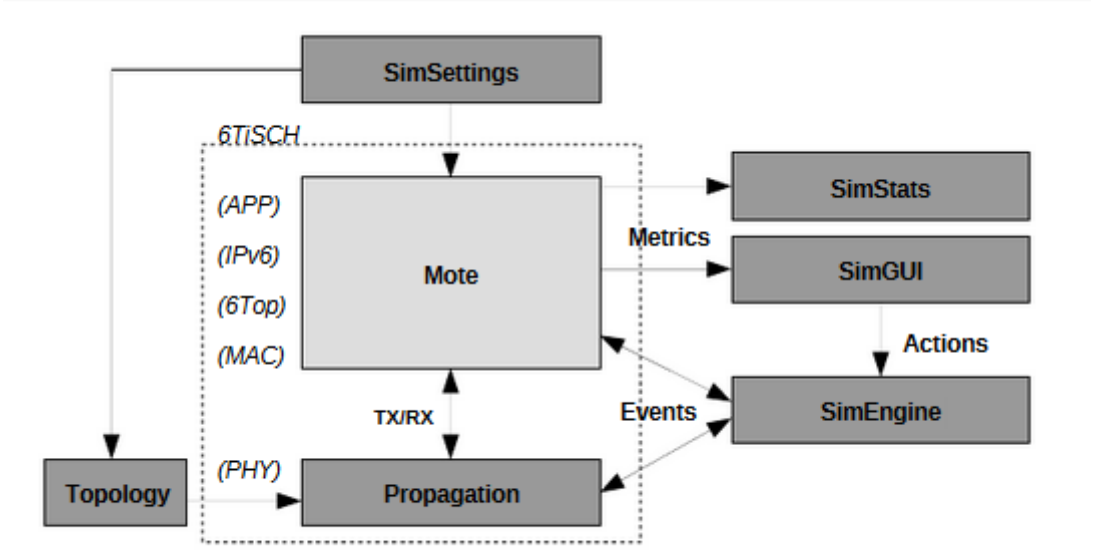


Figure 4.1: Internal architecture of the 6TiSCH simulator retrieved from [3]

4.2.1 SimEngine

SimEngine contains implementation of the event driven core. *Mote* generate events every time a task need to be scheduled. Example of such events are increasing Absolute Slot Number (ASN) or propagating a packet. ASN starts at the first slot, and describes number of slots since the network started. Events are identified by a unique tag consisting of mote ID and Universal Unique Identifier (UUID) of the event. Events are registered with priority as more than one event can happen at the same time instant. Next, the set of events scheduled in the future, Future Event Set (FES), are implemented using a Python list. Events are removed with "pop intermediate" method and added with "insert" method.

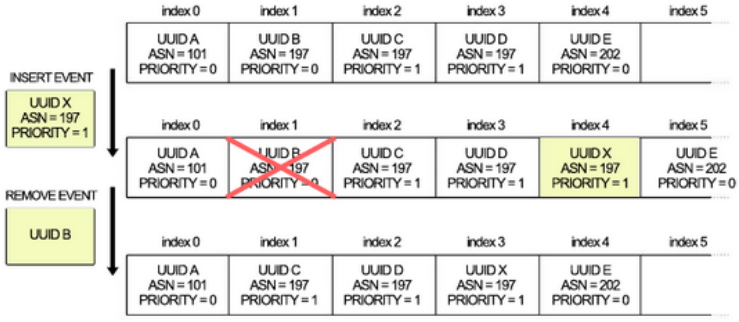


Figure 4.2: FES management example retrived from [3]

Instances in the 6TiSCH simulator has no concurrency risks as they run sequentially in a single thread. Events with lower ASN are always executed first independent of their priority. *SimEngine* prevents events to be inserted into the current ASN in order to remove inconsistencies when popping the next event. Figure 4.2 illustrates how FES manages in the 6TiSCH simulator.

4.2.2 Topology

The simulator generates by default a random topology for each run to make sure network performance are not impacted by specific topology. Deployment area and size can be specified by the user. However, topology can be specified by the user if some modifications are made. When generating the network, motes are by default placed at random until they match the pre-configured mote density. This can be altered as seen in Section 5.1. Motes are not deployed unless each mote has a stable link to a specified minimum number of neighbours. Furthermore, links are assigned a PDR each. Moreover, in *Topology* there are additional configurable topology such as for instance a linear topology.

4.2.3 Propagation model

6TiSCH simulator utilizes a propagation model based on the Pister-Hack model [44]. Pister-Hack model is used to retrieve initial Received Signal Strength Indication (RSSI) value between each pair of motes.

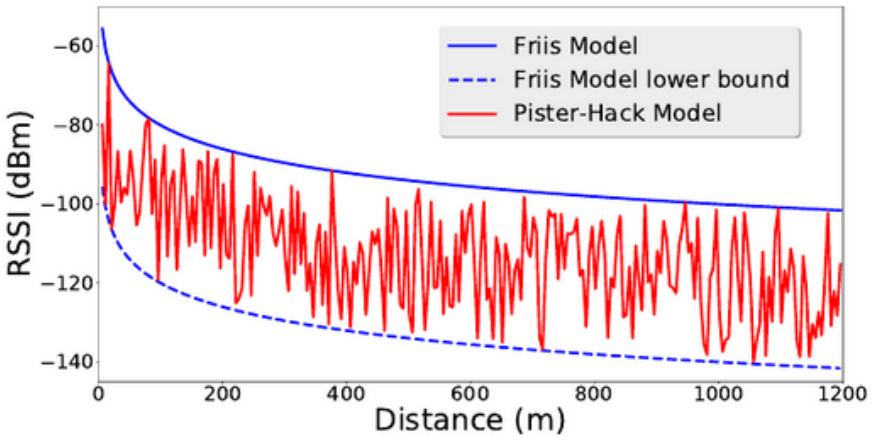


Figure 4.3: The Pister-Hack model generated RSSI values

Next, RSSI values are converted to PDR values through a conversion table. The table reflects relationship between RSSI and PDR accurately in large indoors industrial deployments at the 2.4 GHz band [3]. Pister-Hack model is illustrated in Figure 4.3.

RSSI	PDR
-97 dBm	0.0000
-96 dBm	0.1494
-95 dBm	0.2340
-94 dBm	0.4071
-93 dBm	0.6359
-92 dBm	0.6866
-91 dBm	0.7476
-90 dBm	0.8603
-89 dBm	0.8702
-88 dBm	0.9324
-87 dBm	0.9427
-86 dBm	0.9562
-85 dBm	0.9611
-84 dBm	0.9739
-83 dBm	0.9745
-82 dBm	0.9844
-81 dBm	0.9854
-80 dBm	0.9903
-79 dBm	1.0000

Table 4.2: RSSI to PDR values derived from [3]

A transmission is deemed successful or not by PDR values. Furthermore, in real life wireless scenarios there is interference, and the RSSI from an interfering neighbour transmission is added to the noise to calculate Signal-to-Interference-Plus-Noise Ratio (SINR). Relationship between RSSI and PDR are shown in Table 4.2

4.2.4 Energy consumption model

The simulator implements an energy consumption model based on [2]. This model is briefly presented in Subsection 2.9.5. Different types of slots are defined and combined according to the schedule configuration to find the energy consumption.

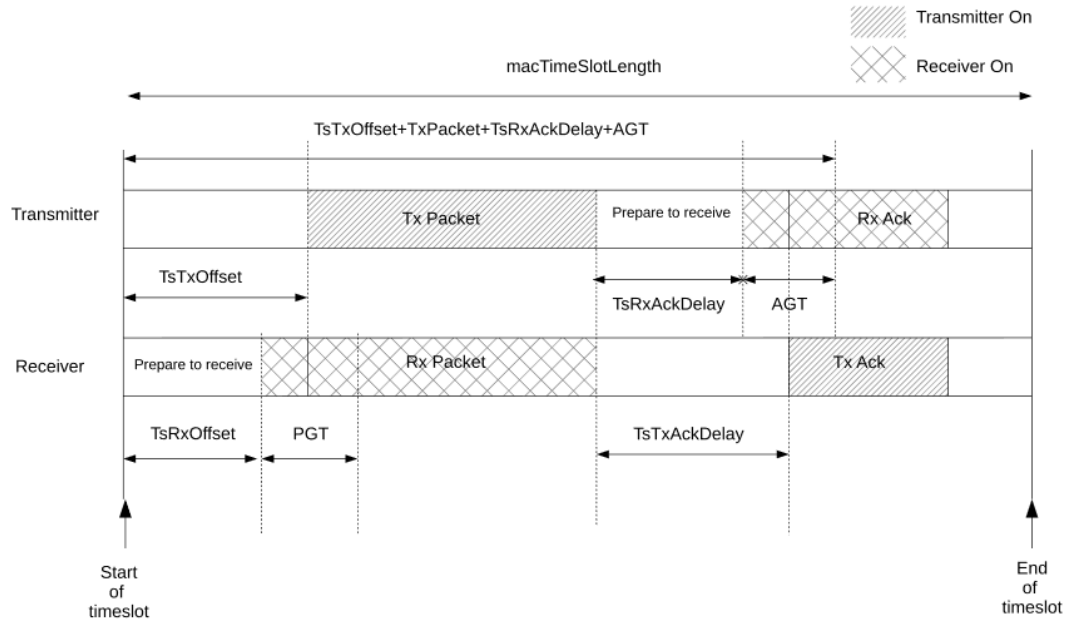


Figure 4.4: The sequence of actions and time-slot timing retrived from [2]

In Figure 4.4 events of a TSCH timeslot are presented. These events are utilized to calculate energy consumption of a mote. An active slot sending a packet and receiving an acknowledgement activates the radio twice: once for transmitting data and once for receiving acknowledgement. Thus, the Central Processing Unit (CPU) is turned on through these phases, while the mote is in a deep sleep mode during other phases. The model defines consumption of sub-periods within the slot and the current draw of the radio when it is active. Other factors are number of bytes transmitted, data rate, and consumption of the CPU in different states and transitions.

State	**Description**
Idle	Mote idle listens. This is a RX state where nothing is received. Hence it only listens for duration of guard time.
Sleep	Mote Deep Sleeps. Slot is off, no CPU nor radio activity due to communication.
TxDataRxAck	Mote transmits a packet and receives an ACK for it.
TxData	Mote sends a broadcast packet not requiring ACK.
RxDataTxAck	Mote receives a packet and responds with an ACK.
RxData	Mote receives a frame that do not require to be acknowledged.

Table 4.3: Different states affecting energy consumption retrieved from [2]

The different states are described in Table 4.3 and after an execution of a

slot the simulator aggregates consumed energy. Lastly, current draw matches the OpenMote platform [45], but can be adapted to other platforms.

4.2.5 Mote

Mote implements the different layers of the 6TiSCH stack. *SimEngine* initiates at boot time a number of *Mote* objects and assigns one of them as root. Next, root adds periodically Enhanced Beacon (EB) and DIOs to its queue to trigger network formation. At the same time, other motes select a random channel and start listening. *Propagation* evaluates at each slot which motes listen and which motes have scheduled transmissions. For every packet the outcome of the transmission in regards to interference and signal strength is determined. Interference level depend on mote density. Each mote sets up TX and RX events according to their schedule. What is more, motes not joined the network yet schedule RX events for waking up at each ASN.

In time, a mote eventually get an EB and synchronize its TSCH MAC layer and start the joining process. Motes joins the network through their Join Proxy which allows it to decipher DIO messages to obtain a rank and set a preferred parent. When preferred parent is selected it triggers dedicated cell allocation. For a mote to be able to send any data traffic, it needs dedicated cells.

Several Housekeeping callback functions are periodically scheduled to perform specific actions at each layer (TSCH, RPL, etc.) to help triggering the booting sequence. In addition, when a timeout is required different events are scheduled (6P timeout, Join timeout, etc.). All events are noted in *SimEngine* and set off the callback functions when the timers expire.

Moreover, all layers in *Mote* can be configured through *SimSettings*. For instance, TSCH layer frame size, timeslot duration and beacon period are examples of configurable parameters. Next, 6top and Minimum Scheduling Function (MSF)[46] can be changed and is currently holding parameters from the original draft. Moreover, RPL is in non-storing mode and support different configurations such as DIO period and DAO period. Lastly, data traffic can be sent at anytime during simulations and can be variable or constant. Traffic bursts can be scheduled and variable traffic is modeled after different probability distributions.

4.2.6 Metrics

As mentioned earlier, the event handler trigger updates for different metrics during the simulation. Currently, over 50 metrics are added and it is possible to add more

if needed. They can be set as *per cycle* to get a values from each slotframe iteration or as an *absolute* to get the total value when the simulation is done. A slotframe cycle is the number of slots in a slotframe, which is by default 101. In Figure 4.4 a list of metrics are shown.

State	Type	**Description**
Average latency	Per cycle	Average latency of packets arriving at root (in ASNs)
Charge consumed	Absolute	Charge consumed by all motes during simulations
Charge consumed at every mote	Absolute	Total charge consumed by a mote during simulations
App packets generated/received	Per cycle	Number of data packets generated and received at every cycle
Number of TX/RX	Per cycle	Number of MAC frames sent and received at every cycle
Number of drops	Per cycle	Drops are classified by its cause: QueueFull, MaxRetries and NoRoute

Table 4.4: Example of available metrics in the simulator derived from [3]

By default, the simulator logs metrics of each simulation in a log file. Moreover, runs utilize different files and folders with regards to CPU ID and network size. Metrics can be plotted from a set of helper scripts designed to process data.

Chapter 5

Simulator implementation

This Chapter describe implementation of Reverse Packet Elimination algorithm in the 6TiSCH simulator. It needed substantial change and new functionality to support RPE. Major code changes made in the 6TiSCH simulator can be located in Appendix C.

5.1 Topology

6TiSCH simulator did not contain capability to specify positions, and thereby a new class named ConnectivitySpecified(ConnectivityBase) was introduced. This class deploys motes in (X, Y) coordinates as show in Listing 5.1. Topology is depicted in Figure 3.3. In other words, 8 motes are deployed with Mote 7 being source and Mote 0 being sink (root).

```
if target_mote.id == 0:
    self.coordinates[target_mote.id] = (0, 0)
    mote_is_deployed = True
```

Listing 5.1: Deploying motes in a (X, Y) grid.

Moreover, to be able to adjust specific PDR of all links a second class ConnectivityMaster ↪ (ConnectivityBase) was created. This class can manipulate the PDR of any link. For the former class, PDR is determined by the RSSI, which in return depend on distance. However, the new class has no concept of positions as PDR is determined by user input as shown in Listing 5.2.

```
connectivity = self.CONNECTIVITY_MATRIX_50_LINK

if  source.id == 0 and destination.id == 1 or source.id == 1 and destination.id
    ↪ == 0:
    connectivity = copy.copy(self.CONNECTIVITY_MATRIX_90_LINK)
```

Listing 5.2: Specifying PDR of selected links and setting a default value for all others

5.2 Path computation element

Firstly, to achieve deterministic behavior for data packets, a fairly simple PCE is implemented in *Mote*. This PCE work on top of MSF [46] with purpose of introducing 6TiSCH tracks for Path A and Path B from source to sink. Tracks are implemented from source towards sink, and in reverse from sink to source shown in Figure 5.1 and listed underneath:

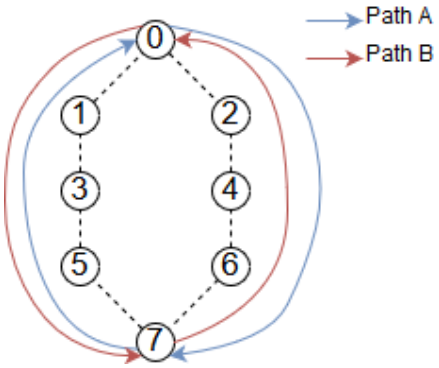


Figure 5.1: Topology PCE schedules tracks for. Red arrow indicates Path B and blue indicates Path A

- Path A: Source → Mote 5 → Mote 3 → Mote 1 → Sink
- Reverse Path A: Source ← Mote 5 ← Mote 3 ← Mote 1 ← Sink
- Path B: Source → Mote 6 → Mote 4 → Mote 2 → Sink
- Reverse Path B: Source ← Mote 6 ← Mote 4 ← Mote 2 ← Sink

PCE schedules TSCH cells in daisy chains for all paths. Path A is scheduled to send first. Path B however, is as mentioned in Chapter 3, scheduled with a delay τ. Daisy chain delay is configurable as shown in Listing 5.3.

```
#INDEX IS FROM 0-100 (101)
self.slotA = 1
self.slotB = 2 #ADJUST THE DAISY CHAIN DELAY
self.slotReverseA = 6
self.slotReverseB = 5 #DAISY CHAIN DELAY + 5
```

Listing 5.3: Adjusting the packet replication delay

TSCH forward data packets in these cells, and no other traffic are permitted to utilize these tracks. PCE set tracks upwards to sink and downwards from sink. Next, it schedule TX and RX cells at a random channel as shown in Listing 5.4

```
for idx, moteid in enumerate(path):
    self.slotA += 1
    channel = randint(0, 3)
    if idx >= len(path) - 1:
     break
    sender = self.engine.motes[moteid]
    receiver = self.engine.motes[path[idx + 1]]

    sender.tsch.addCell(
        slotOffset       = self.slotA,
        channelOffset    = channel,
        neighbor         = receiver.get_mac_addr(),
        cellOptions      = [ d.CELLOPTION_TX ],
        trackID          = 0,
        slotframe_handle = 0,
    )
    receiver.tsch.addCell(
        slotOffset       = self.slotA,
        channelOffset    = channel,
        neighbor         = sender.get_mac_addr(),
        cellOptions      = [ d.CELLOPTION_RX ],
        trackID          = 0,
        slotframe_handle = 0
    )
```

Listing 5.4: Scheduling TX and RX cells for Path A

6TiSCH tracks are deployed at ASN 1 and do not allow any other traffic than data generated at App layer. Reverse paths are a continuation of the daisy chains from Path A and Path B, but traffic is only sent at these tracks if triggered by sink. In fact, it is at the reverse tracks RPE packets operate.

5.3 TSCH

At the TSCH layer a track ID is introduced to ensure only traffic for a specific track is scheduled at that particular track. If a packet needs to be retransmitted it has to wait a slotframe iteration for a new opportunity as it is not allowed to use any other TX cells. This is true for other traffic, a packet will not be put in the TX queue for a cell that is owned by a track even if the track has nothing scheduled. This ensures no random behaviour and enables deterministic tracks. This is illustrated in Listing 5.5

```
types = [d.PKT_TYPE_PATH_A, d.PKT_TYPE_PATH_B, d.PKT_TYPE_RPE_A, d.PKT_TYPE_RPE_B]
for packet in self.txQueue:
    if packet['mac']['dstMac'] == dst_mac_addr:
        if trackID is not None:
            if packet['type'] in types:
                packet_to_send = packet
                break
        else:
            if packet['type'] not in types:
                packet_to_send = packet
                break
```

Listing 5.5: Ensuring no randomness in 6TiSCH tracks

5.4 Scheduling function

The 6TiSCH simulator deploys MSF by default. However, to ensure normal operation of MSF do not interfere with 6TiSCH tracks a few modifications need to be applied. As mentioned tracks are introduced at ASN 1, so to ensure MSF do not schedule traffic in these cells some checks need to be done. Firstly, getting cells available in the intersection of slotframe handles as shown in Listing 5.6

```
available_slots  = list(
    slots_in_cell_list.intersection(
        set(self.mote.tsch.get_available_slots(self.SLOTFRAME_HANDLE)) -
        self.locked_slots
    )
)
```

Listing 5.6: Getting available cells from slotframe handles

Next, track cells are statically configured in Listing 5.7. Cells are removed from the pool of cells MSF has available to ensure no cells are double booked.

```
def _get_available_slots_global(self):
    busyslots = []
    for mote in self.engine.motes:
        for key in mote.tsch.slotframes:
            slots = mote.tsch.slotframes[key].get_busy_slots()
            for slot in slots:
                busyslots.append(slot)
    return list((set(range(self.engine.settings.tsch_slotframeLength))) - set(
    ↪ busyslots))
```

Listing 5.7: Retrieving available cells

Finally, MSF operate without utilizing cells configured for tracks. This is shown for RX cells in Listing 5.8, TX cells corresponds with scheduled RX cells.

```
def _get_autonomous_cell(self, mac_addr):
    return self.engine.get_mote_by_mac_addr(mac_addr).sf.autonomous_cell

def _allocate_autonomous_rx_cell(self):
    all_slots = self._get_available_slots_global()
    selected_slot = random.choice(all_slots)
    channel_offset = random.randint(0, 15)
    self.mote.tsch.addCell(
        slotOffset        = selected_slot,
        channelOffset     = channel_offset,
        neighbor          = None,
        cellOptions       = [d.CELLOPTION_RX],
        slotframe_handle = self.SLOTFRAME_HANDLE
    )
    self.autonomous_cell = (selected_slot, channel_offset)
```

Listing 5.8: Scheduling cells without interfering with tracks

5.5 RPL

RPL layer is modified to set static routes upstream. Motes should normally find their parent, but to ensure explicit routes parent selection is predefined. Next, as the implementation requires source to have two parents to ensure the copies are sent along disjoint paths, Mote 7 will before sending a packet switch parent. Switch is done at *App*. Parents selection is shown in Listing 5.9

```
if self.preferred_parent is None and self.rpl.mote.id != 0:
    parent_mote = None
if self.rpl.mote.id == 1:
    parent_mote = self.rpl.engine.motes[0]
elif self.rpl.mote.id == 2:
    parent_mote = self.rpl.engine.motes[0]
```

Listing 5.9: Influencing parent selection of motes. In this example Mote 1 and Mote 2 sets sink as their preferred parent

5.6 Application layer

Application Layer (APP) resides at each mote. All motes operate the same logic, except the sink. To ensure simulations are not affected by any other traffic, only sink and source are allowed to generate data traffic. At source, traffic is only generated after a user specified number of slotframe iterations to ensure the network is converged. Next, users can set how often data is sent. For instance, every 10 slotframes source generates a packet for each path. Number of packets sent for each run is configurable. These settings can be changed as shown in Listing 5.10.

```
waitSlotframes = 10
slotframe = 101
converged = 10000
sendpackets = 4000
```

Listing 5.10: Configurable parameters at Source

Next, as mentioned in Section 5.5, source needs to alternate between parents. This is done by setting preferred parent before sending each packet as illustrated in Listing 5.11.

```
if  self.mote.id == 7 and self.engine.asn > converged:
    isNewSlotFrame = (self.engine.asn % (slotframe*waitSlotframes)) == 0
    if isNewSlotFrame:

        self.engine.motes[7].rpl.of.set_preferred_parent(self.engine.motes[5].
    ↪ get_mac_addr())

        self._send_path_a(
        dstIp          = self.mote.rpl.dodagId,
        packet_length  = self.settings.app_pkLength
        )
```

```
        self.engine.motes[7].rpl.of.set_preferred_parent(self.engine.motes[6].
    ↪ get_mac_addr())

        self._send_path_b(
        dstIp          = self.mote.rpl.dodagId,
        packet_length  = self.settings.app_pkLength
        )
```

Listing 5.11: Scheduling data packets and switching parents before doing so

Packets then propagate their paths towards the sink. When a packet arrive at sink, it does a similar check as Listing 5.17 to check if it has a packet with the sequence number in its TSCH queue. If the sink has a copy in its queue the packet is dropped, if not a RPE packet is triggered down the opposite path through Listing 5.12.

```
if not foundAck:
    if packet['type'] == d.PKT_TYPE_PATH_A and packet['mac']['srcMac'] == self.
    ↪ engine.motes[1].get_mac_addr():
        self._send_rpe_b(packet['app']['sequencenumber'])

    elif packet['type'] == d.PKT_TYPE_PATH_B and packet['mac']['srcMac'] == self.
    ↪ engine.motes[2].get_mac_addr():
        self._send_rpe_a(packet['app']['sequencenumber'])
```

Listing 5.12: Triggering the RPE packet downstream from sink

5.6.1 Increasing delay beyond slotframe

Major changes to the logic had to be made to enable the simulator to schedule a packet into the future. In essence, in terms of sending packets there are no reasons to schedule a packet and withhold it for a number of slotframes before sending it. However, this was solved by utilizing *SimEngine* function self.engine.scheduleAtAsn enabling the simulator to schedule an event into the future. This is presented in Listing 5.13, as illustrated Path A packet is scheduled now and Path B packet simultaneously is scheduled into the future.

```
    if   self.engine.asn % 1010 == 0:
        self.engine.motes[7].rpl.of.set_preferred_parent(self.engine.motes[5].
    ↪ get_mac_addr())
        self._send_path_a(
            dstIp          = self.mote.rpl.dodagId,
            packet_length  = self.settings.app_pkLength
        )
        self.txpacket += 1
        self.engine.scheduleAtAsn(
            asn            = self.engine.getAsn() + (101*16), # number of
    ↪ slotsframes to wait
            cb             = self._sendB,
            uniqueTag      = "criticalpacket_b" + str(self.sequencenumber),
            intraSlotOrder = d.INTRASLOTORDER_ADMINTASKS
        )
```

Listing 5.13: Scheduling an event into the future

Next, in this example Path B is scheduled 16 slotframes into the future, and need to be de-scheduled in the event of an RPE packet looping back. In addition, sequence numbers need to match, presenting an issue as sequence numbers are incremented every 10 slotframes. An easy fix is done by decrementing sequence number of Path B packet in the future before putting it in the TSCH queue. Moreover, de-scheduling method is presented in Listing 5.14

```
def recvPacket(self, packet):
    if self.mote.id == 7 and packet['type'] == d.PKT_TYPE_RPE_B:
        print 'PATH A Returned'
        sequencenumber = packet['app']['sequencenumber']
        self.engine.removeFutureEvent("criticalpacket_b" + str(sequencenumber))
        self.mote.drop_packet(
                            packet  = packet,
                            reason  = SimEngine.SimLog.REVERSE_DROP,
        )
    else:
        pass
```

Listing 5.14: De-scheduling an future event

What is more, an issue with this approach, or in fact a drawback with the simulator is logging of latency. When a packet is received and latency is calculated, it looks at the first slot of the slotframe as beginning of time. Meaning, if a packet is scheduled at slot 4 and received at slot 8, the latency is 80 milliseconds and not 40 milliseconds. This is not an issue in itself, but when utilizing self.engine.scheduleAtAsn it presents a problem. The problem is the event is scheduled into the future, meaning a packet has no concept of time before that particular ASN. Consider the same scenario with a packet arriving at slot 8 after 16 slotframes, in terms of latency this would still show up as 80 milliseconds, as the packet do not know 16 seconds has passed. Meaning, effect of an increased sending delay would not appear in the results. However, as results are calculated after the simulation is done by iterating through a log file, a correction can be made. As shown in Listing 5.15 latency is altered by adding 16 slotframes to the calculation every time a Path B packet is received at sink. This method works, but is not exactly "by the books".

```
if logline['packet']['type'] == d.PKT_TYPE_PATH_B:
                allstats[run_id][mote_id]['upstream_pkts'][appcounter]['rx_asn'] =
    ↪  asn + (101*16)#Adjust delay to logfile
            else:
                allstats[run_id][mote_id]['upstream_pkts'][appcounter]['rx_asn'] =
    ↪  asn
```

Listing 5.15: Correcting delay before logging latency

5.7 6LoWPAN

Next, to assure packets are sent where they are supposed to, static forwarding routes downstream for RPE packets are implemented at 6LoWPAN. Ideally, this

should be done by a routing protocol, but routing protocols are not subject of analysis. Downstream routes only allow specific traffic types as shown in Listing 5.16

```
if packet['type'] == 'RPE_A' and self.mote.id == 0:
    return self.engine.motes[1].get_mac_addr()

if packet['type'] == 'RPE_B' and self.mote.id == 0:
    return self.engine.motes[2].get_mac_addr()
```

Listing 5.16: Setting the downstream paths in 6LoWPAN. In this listing Sink sets Mote 1 and Mote 2 as receivers of RPE A and RPE B

Moreover, 6LoWPAN is modified to allow new packet types, and it is at this layer the drop mechanism RPE utilizes is implemented. In essence, RPE triggers a mote to check its TSCH queue. This check is done 3 times:

- When receiving a packet
- When sending a packet
- When forwarding a packet

A check is done this often to assure upstream packets are dropped. For instance, when a packet is received it checks it queue for the packet, and if its not there the packet is pushed to the forwarding layer. But, when the packet is at the forwarding layer an upstream packet may be received, and if the check is not done again it may pass by undetected. Dropping a upstream packet method is presented in Listing 5.17:

```
if packet['type'] == d.PKT_TYPE_RPE_A or packet['type'] == d.PKT_TYPE_RPE_B:
    for Qpacket in self.mote.tsch.txQueue:
        if Qpacket['type'] == d.PKT_TYPE_PATH_A or Qpacket['type'] == d.
    ↪ PKT_TYPE_PATH_B:
            if packet['app']['sequencenumber'] == Qpacket['app']['sequencenumber'
    ↪ ]:
                self.mote.tsch.dequeue(Qpacket)
                print 'dropped at', self.mote.id, 'Packet', Qpacket['type']
                self.mote.drop_packet(
                        packet   = Qpacket,
                        reason   = SimEngine.SimLog.REVERSE_DROP,
                )

                goOn = False
```

Listing 5.17: Dropping packets going upstream

The same method need to be modified and applied to sink. Ensuring if a packet arrives at sink and a packet from the other path arrives before a RPE packet is sent, it is dropped before reaching APP layer. If Listing 5.18 check is not done, sink registers two packets as arrived and note both latencies.

```
if self.mote.id == 0:
    if packet['type'] == d.PKT_TYPE_PATH_A or packet['type'] == d.PKT_TYPE_PATH_B:
        for Qpacket in self.mote.tsch.txQueue:
            if Qpacket['type'] == d.PKT_TYPE_RPE_A or Qpacket['type'] == d.
    ↪ PKT_TYPE_RPE_B:
                if packet['app']['sequencenumber'] == Qpacket['app']['
    ↪ sequencenumber']:
                    self.mote.tsch.dequeue(Qpacket)
                    self.mote.tsch.dequeue(packet)
                    print 'Dropped at 6lowpan at Sink:' ,packet['type']
                    goOn = False
```

Listing 5.18: Dropping packets before they enter the APP layer

5.8 Mote

At *Mote* a change was added allowing the network to be synchronized at TSCH level at the start of each run. In essence, motes need synchronize their clocks to ensure they are listening and sending when they are supposed to. Furthermore, to ensure this all motes need predefined join proxies and knowledge of what clock to sync against. Setting these are illustrated in Listing 5.19. To summarize, This is done to decrease simulation time, and to ensure results have identical conditions.

```
        self.secjoin.setIsJoined(True)
        # tsch
        if self.id == 1:
            self.tsch.join_proxy =  netaddr.EUI(self.engine.motes[0].get_mac_addr
    ↪ ())
            self.tsch.clock.sync(self.engine.motes[0].get_mac_addr())
```

Listing 5.19: Syncing TSCH clock and setting join proxy at *Mote*.

5.9 General simulator setup

This section present general parameters utilized. All parameters are derived from [39]. These are current best practice of 6TiSCH, but not standardized. However, some of the parameters such as packet interval, number of packets and packet size are not listed as they are application specific. Moreover, all parameters are identical for all runs, with exception of link quality and delay τ. However, these are specified with corresponding results to insure all results are interpreted properly.

Description	Value
Slotframes per run #	21000
Packet size	127 bytes
DAO period	60 seconds
Scheduling function	Minimal scheduling function
Slot duration	0.01 seconds
Slotframe length	101 slots
Enhanced beacon probability	0.16
Clock max drift part per million	30 ppm
Keep alive interval	10 seconds
Number of physical channels #	16
Number of packets #	2000
Packet interval	1 Packet every 10 slotframe
Link quality	Specified for each run
Delay τ	Specified for each run
Maximum retransmissions #	4
Number of runs #	30

Table 5.1: General parameters of simulator

5.9.1 Slot charge

Simulations has utilized predefined values set in the 6TiSCH simulator in terms of slot charges. These values are listed in Table 5.2.

State	Coloumbs
Idle	$6.4\mu C$
TxDataRxAck	$54.5\mu C$
TxData	$49.5\mu C$
RxDataTxAck	$32.6\mu C$
RxData	$22.6\mu C$

Table 5.2: Energy charges in terms of consumption

5.9.2 Validation of results

As shown in Table 5.1 number of runs are 30. Moreover, standard deviation was used to calculate Coefficient of variation (CV), known as relative standard deviation. Highest CVs recorded are listed underneath:

- Battery life: 8%

- Packet delivery ratio: 0.04%
- Average latency: 4%
- 99th percentile: 6%
- Packet elimination location: 12%

5.9.3 Hardware

Time spent simulating different setups depend on CPU speed and number of cores. Each run lasts approximately 8 minutes when utilizing hardware shown in Table 5.3.

Description	**Value**
Intel core i5-8600K	Cores: 6 / Threads: 6
Processor frequency (max)	3.6GHz (4.3 GHz)
RAM	2x8GB DDR4 SDRAM

Table 5.3: Hardware utilized to simulate results

5.9.4 Limitations

As shown in Section 5.5, preferred parent selection is influenced and changes every time source alternates between Path A and Path B. Selection of different parents, namely DP and Alternative Parent (AP), is in itself a field to be studied. Issues such as how motes should distribute their DAO messages. This problem is assessed in IETF draft [47] published in January 2019. Moreover, a small amount of the simulations, 1 or 2 motes do not join the DODAG properly. These results have been eliminated.

5.10 Scenarios

This section present simulated scenarios. Main simulation setups are presented before parameters utilized are discussed.

5.10.1 Topology

All simulations follow the topology depicted numerous times. But, to ensure no misunderstandings a brief recap is presented with changes in naming convention.

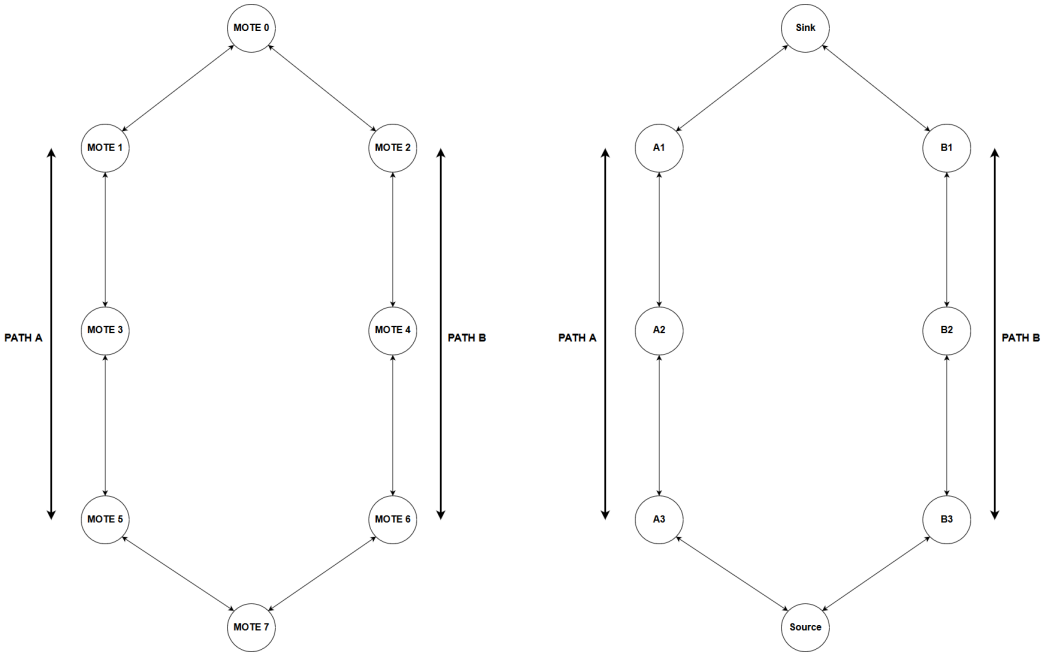


Figure 5.2: Changing the naming convention

Changes are illustrated in Figure 5.2, where Figure (a) represents previous naming convention, and Figure (b) depicts the change. In essence, names are changed from mote number, to path location and number. For example, Mote 1 is now A1. This is to get a better understanding of upcoming results.

5.10.2 Single Path

When referring to Single Path, it describes a scenario where traffic is only sent from source to sink through Path A. No data traffic except normal 6TiSCH operations happens at Path B. This is assessed as normal 6TiSCH operation. There are no packet replication or elimination involved. In some figures Single Path is referred to as SP. Single Path can be visualized in Figure 5.3.

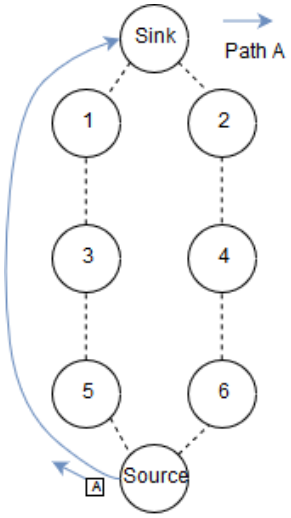


Figure 5.3: Illustration of Single Path scenario

5.10.3 Dual Path

Dual Path refers to a scenario where there is packet replication at source and packet elimination at sink. This version contains the packet elimination and replication technique presented in DetNet [17] and analyzed in [5]. In some figures Dual Path is referred to as DP.

5.10.4 RPE

RPE refers to implementation of the novel Reverse Packet Elimination proposal. This scenario is listed with corresponding sending delay τ in figures.

5.10.5 RPE Overprovisioning

In this scenario PCE schedules an extra TX and RX slot for each upstream hop. Otherwise, this refers to RPE operation with $\tau = 8$. In graphs RPE Overprovisioning is referred to as OP.

5.10.6 Parameters

Table 5.4 list PDR utilized at the links. Next, Table 5.5 shows the sending delay τ

PDR	70%	80%	90%

Table 5.4: Link qualities utilized in simulations

τ	1	8	816	1624

Table 5.5: Different parameters in term of sending delay

5.10.7 6TiSCH tracks

In all simulations, tracks are implemented from source to sink and in return. This is done as DetNet request a track for each mote. In Single Path version, this is only done at Path A.

Chapter 6

Results

This chapter present all results, starting with theoretical results. Next, simulation results are presented with regards to latencies, reliability and mote lifetimes.

6.1 Theoretical results

This section shows analytical results. Firstly, taking a look at theoretical latencies in terms of maximum and minimum. Next, show packet size influences radio up time. Then, how packet size effect expected number of transmissions over multiple hops. Finally, expected mote lifetime with regards to packet size.

6.1.1 Maximum and minimum latencies

Given topology in Figure 5.2 with 4 hops from source to sink, theoretical maximum and minimum latencies with given τ are listed in Table 6.1.

τ	**Latency minimum** [sec]	**Latency maximum** [sec]
0	0.04	16.16
1	0.04	16.17
8	0.04	16.24
816	0.04	24.32
1624	0.04	32.40
OP	0.07	16.28

Table 6.1: Maximum and minimum theoretical latencies for τ in seconds

What Table 6.1 show is increasing delay τ increases theoretical maximum latency. With $\tau = 0$ it indicates Single Path with no packet replication. Moreover,

increasing τ from 8 to 1624 nearly doubles theoretical maximum from 16.24 seconds to 32.40 seconds. What is more, all scenarios have minimum latency at 40 milliseconds, except Overprovisioning due to extra cells in tracks. Hence, Overprovisioning have 70 milliseconds as lowest possible latency.

6.1.2 Radio up time

Time required for the radio to be active sending 25 packets with max packet size 127 bytes and 25 RPE packets with a size of 23 bytes are shown in Figure 6.1. Furthermore, ACK is taken into account. This scenario assumes perfect conditions (e.g no loss).

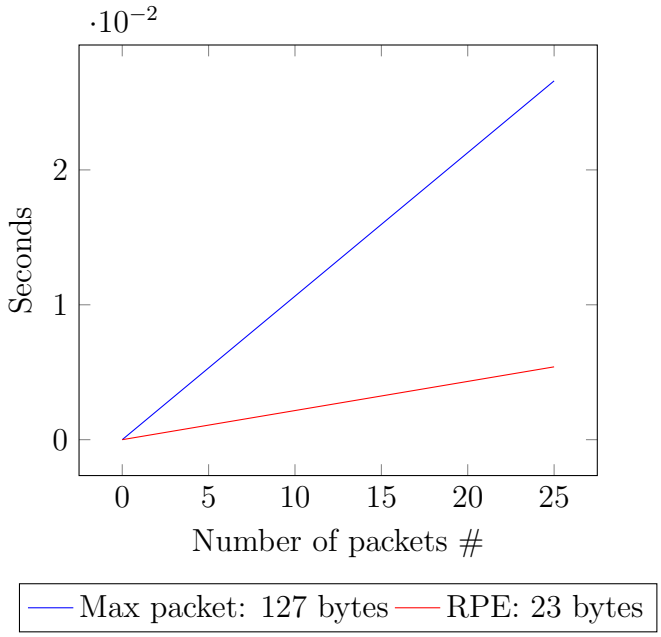


Figure 6.1: Time to transmit a max packet and a RPE packet assuming no loss

Figure 6.1 show after 25 packets are transmitted radio up time is reduced from 0.0266 seconds to 0.0054 seconds, a 21.2 milliseconds reduction. To summarize, RPE packets require less radio up time than larger packets.

6.1.3 Radio up time with retransmissions

As Figure 6.1 do not take effects of BER into consideration, it does not give a realistic representation. Thus, Figure 6.2 present expected radio up time for a single packet as a function of BER. Packet sizes are max packet size 127 bytes and RPE at 23 bytes. When the plots flatten packets are assumed to fail as the limit

has reached m, maximum transmission attempts as derived in Subsection 2.8.1. Figure 6.2 shows a 127 bytes packet flattens at approximately 4.25 milliseconds with a BER at 0.005. RPE packet with 23 bytes flattens at roughly 0.7 milliseconds when BER is 0.03.

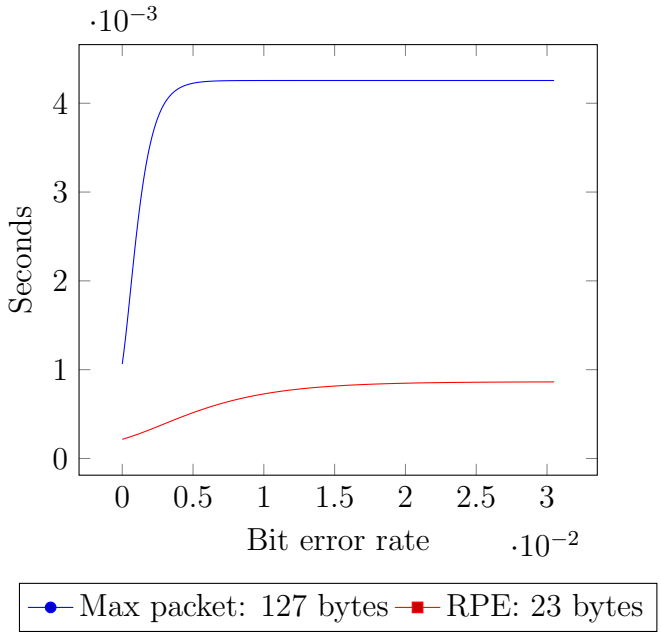


Figure 6.2: Expected radio up time considering BER for a max packet 127 bytes and RPE 23 bytes

Figure 6.1 shows packet size has an impact on radio up time. An increased amount of bits in a packet effect likelihood of a unsuccessful transmission. A max packet size caps at BER 0.5% while a RPE packet go as high as a BER of 3%. What this shows, is that theoretically a RPE packet has a higher chance of being successfully transmitted and utilizes less radio up time compared to a 127 bytes packet.

6.1.4 Expected number of transmissions

Given the topology consisting of 4 hops, expected number of transmissions for PDRs utilized are illustrated in Figure 6.3.

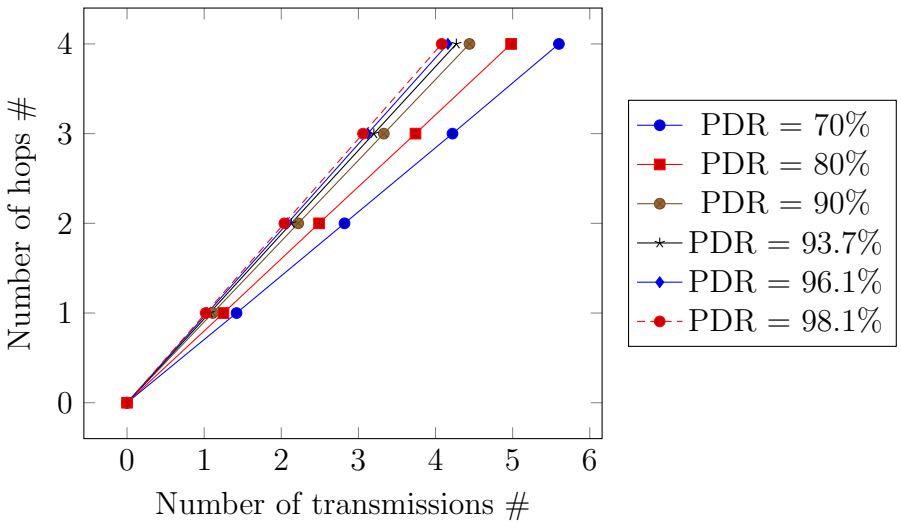


Figure 6.3: Expected number of transmissions with given PDRs

Selected PDRs are based on work in Section 2.6, where Table 2.2 derives relationship between PDR and packet size. What is more, BERs are retrieved from the same section. Assume scenario $\tau = 8$ described in Subsection 3.1.3 and that the first copy, Path A packet, has made it to the sink. Thus, expected number of transmissions a RPE packet requires to make it back to source are shown in Table 6.2.

BER	**PDR**	**Expected number of transmissions**
$3.483 \cdot 10^{-4}$	93.7%	4.27
$2.178 \cdot 10^{-4}$	96.1%	4.16
$1.028 \cdot 10^{-4}$	98.1%	4.08

Table 6.2: Expected number of transmissions for a RPE packet the given PDRs

Next, assume a packet along Path A has failed and packet is transmitted to sink along Path B. Expected number of transmissions for packet along Path B to reach sink are shown in Table 6.3. As Path A == Path B, expected number of transmissions for a packet along Path A are equal to Table 6.3.

BER	**PDR**	**Expected number of transmissions**
$3.483 \cdot 10^{-4}$	70%	5.60
$2.178 \cdot 10^{-4}$	80%	4.98
$1.028 \cdot 10^{-4}$	90%	4.44

Table 6.3: Expected number of transmissions for max packet size with the given PDRs

Table 6.2 and Table 6.3 show expected number of transmissions increases with identical BER due to increased packet size. Emphasizing, PDR changes when increasing packet size for same BER. Max packet size uses 5.60 transmissions and RPE uses 4.27 transmissions with a BER at $3.483 \cdot 10^{-4}$. In essence, this shows a max packet size has a higher chance of utilizing retransmission attempts. Theoretically, RPE packets on their way downstream can stop larger packets stuck retransmitting upstream. For instance, if a link between two motes deteriorate, larger packets utilize more transmission attempts than a smaller packet. Hence, RPE packets has higher chance of successful transmission over the same link, and can stop larger packet from utilizing all their retransmission attempts. In short, this would theoretically decrease energy consumption as larger packets stuck in retransmission upstream can be dropped.

6.1.5 Expected mote lifetime

Next question is how packet size affect expected TSCH mote lifetime. In this analysis everything except 1 TX slot and 1 RX slot are removed, remaining 99 slots are in sleep mode. Emphasizing, all other slots are removed, meaning no minimal slot or shared slots. This is to get a clear view of effects RPE packet size has compared to max packet size without any RPL, TSCH or 6top traffic clouding the results. Packet rate is 1 per slotframe iteration.

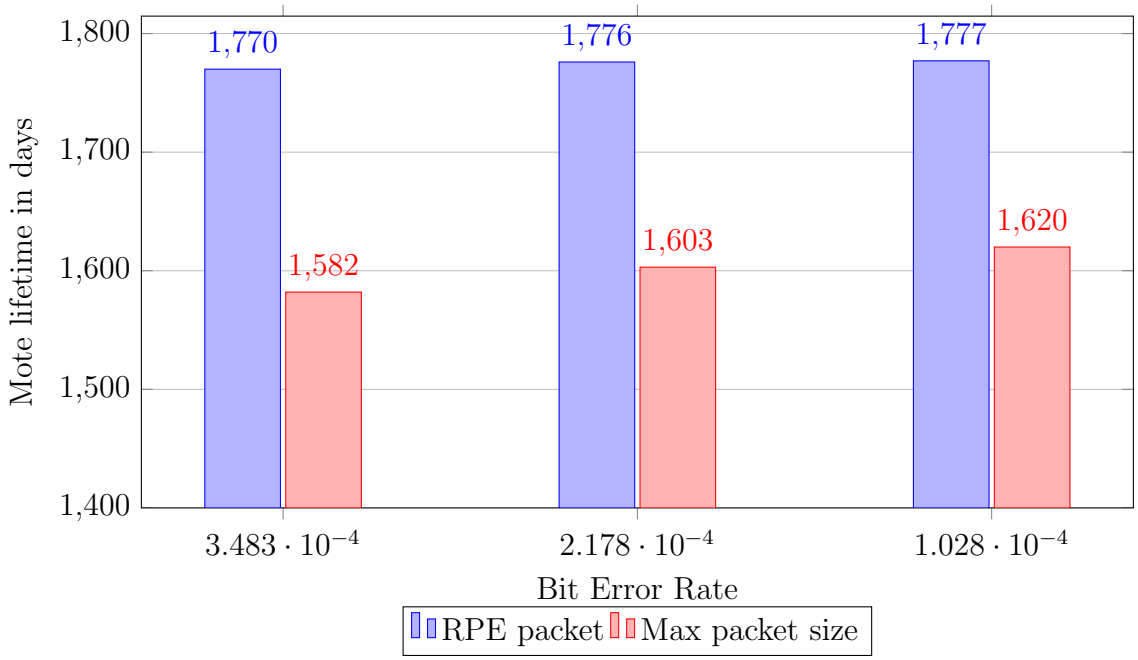


Figure 6.4: Theoretical expected mote lifetimes in days when transmitting max packet size and RPE packets

Using same case as previous, $\tau = 8$, where a packet along Path A has made it to sink. This triggers a RPE packet from sink to source down Path B. Lifetime for each mote with different PDR are then shown in Table 6.4.

PDR	**BER**	**Expected lifetime [years]**	**Expected lifetime [days]**
93.7%	$3.483 \cdot 10^{-4}$	4.84	1770
96.1%	$2.178 \cdot 10^{-4}$	4.86	1776
98.1%	$1.028 \cdot 10^{-4}$	4.0	1777

Table 6.4: Expected TSCH mote lifetime when using RPE packet

Next, as before, assume a packet along Path A fails which leads to a packet being transmitted at Path B. Mote lifetime for each mote transmitting a max packet size are listed in Table 6.5. These results are identical for all motes along Path A.

PDR	**BER**	**Expected lifetime [years]**	**Expected lifetime [days]**
70%	$3.483 \cdot 10^{-4}$	4.33	1582
80%	$2.178 \cdot 10^{-4}$	4.38	1603
90%	$1.028 \cdot 10^{-4}$	4.43	1620

Table 6.5: Expected TSCH mote lifetime when using max size packet

Table 6.6 looks at difference in lifetime for each mote in percent. Moreover, Figure 6.4 illustrates difference in days.

BER	**RPE packet [days]**	**Max packet size [days]**	**Increase [%]**
$3.483 \cdot 10^{-4}$	1770	1582	11.9%
$2.178 \cdot 10^{-4}$	1776	1603	10.8%
$1.028 \cdot 10^{-4}$	1777	1620	9.7 %

Table 6.6: Theoretically increased lifetime for each mote in %

In essence, what this section show is decreasing packet size, increases mote lifetime. Increasing mote lifetimes, decreases maintenance as motes do not need to be recharged or changed as often. In a industrial wireless sensor network this is crucial as downtime can stop production causing revenue loss.

6.1.6 Summarizing theoretical results

Before presenting simulation results, a summary of theoretical results is presented. What this section has shown is: firstly, increasing replication delay τ increases maximum latency. Secondly, in terms of packet size, RPE packet requires less time to be transmitted. Next, it has a higher chance of successfully being transmitted as it tolerates a higher BER. Finally, motes transmitting RPE packets have a lower energy consumption. Moreover, the next sections presents simulated results.

6.2 Reliability

This section present reliability and number of packets lost. As mentioned in Subsection 5.10, RPE scenarios are listed with replication delay τ. The comparison is done between Single Path and $\tau = 8$ as all scenarios utilizing packet replication has a negligible difference in reliability due to identical behaviour.

6.2.1 Packet delivery ratio

As shown in Figure 6.5(a) and Table 6.7 with a link quality of 70%, Single Path has a reliability of 88.24%. RPE increases the reliability at the same link with 10.41% to 98.65%. Moreover, increasing quality of links to 80% decreases difference between RPE and Single Path, but it is still significant with Single Path at 97.7% compared to RPE with 99.95%. Lastly, at 90% link quality the gap in reliability closes in with a difference of 0.09% as RPE delivers a 100% delivery ratio. However, there is packet loss involved, and over time statistically a packet will be dropped.

Hence, delivery ratio cannot be 100% and is presumably in the range of 99.999% as DetNet and TSCH are aiming to achieve [17][16].

PDR	Single Path delivery ratio	RPE delivery ratio
70%	88.24%	98.65%
80%	97.7%	99.95%
90%	99.91%	100%

Table 6.7: Packet delivery ratios in percent with different link qualities

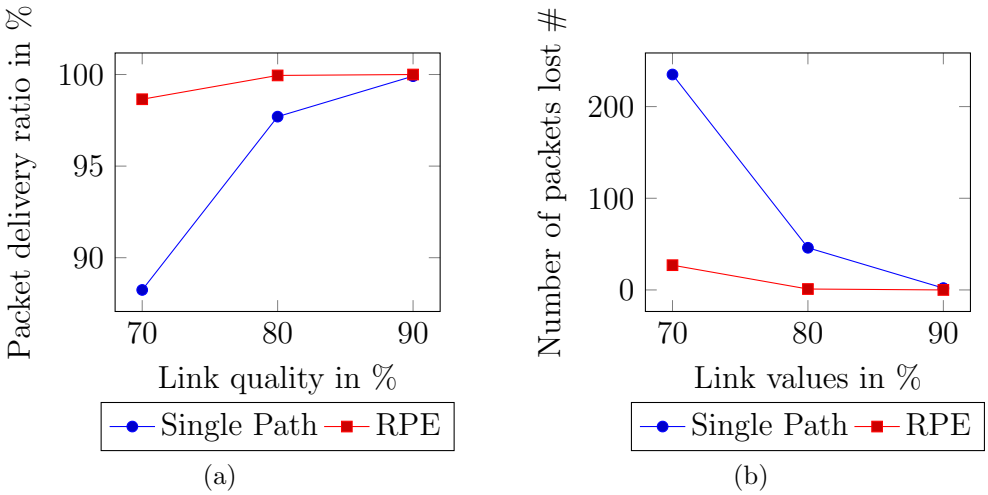


Figure 6.5: Number of packets lost and packet delivery ratios with link quality from 70-90%

Packets lost

In this subsection, the delivery ratios achieved are illustrated with packets dropped. Number of packets lost out of 2000 sent are listed in Table 6.8 and depicted in Figure 6.5(b). For instance, if packet replication is not utilized at 70% link quality, number of packets lost are increased from 27 to 235, a difference of 208 packets. Next, increasing quality of links to 90% closes the gap from 208 to a difference of 2. In an industrial network getting all the data is important and seen in Table 6.7, with 80 link quality reliability is increased with 2.25%. A 2.25% increase in reliability in this example is a reduction in packets lost from 46 to 1. To summarize, a few percent increase in reliability matters as industrial networks require high reliability.

PDR	Single Path - Packets lost #	RPE - Packets lost #
70%	235	27
80%	46	1
90%	2	0

Table 6.8: Number of packet lost when utilizing different link qualities

6.3 Distribution of RPE packets

This section present distribution of where RPE packets eliminated upstream packets along Path A or Path B. This shows the effect of introducing shorter and longer delays τ. Figure 6.6 shows the different PDR values changes elimination location. Particularly, lowering link quality causes elimination location to spread across all motes with a higher rate than when link quality increases.

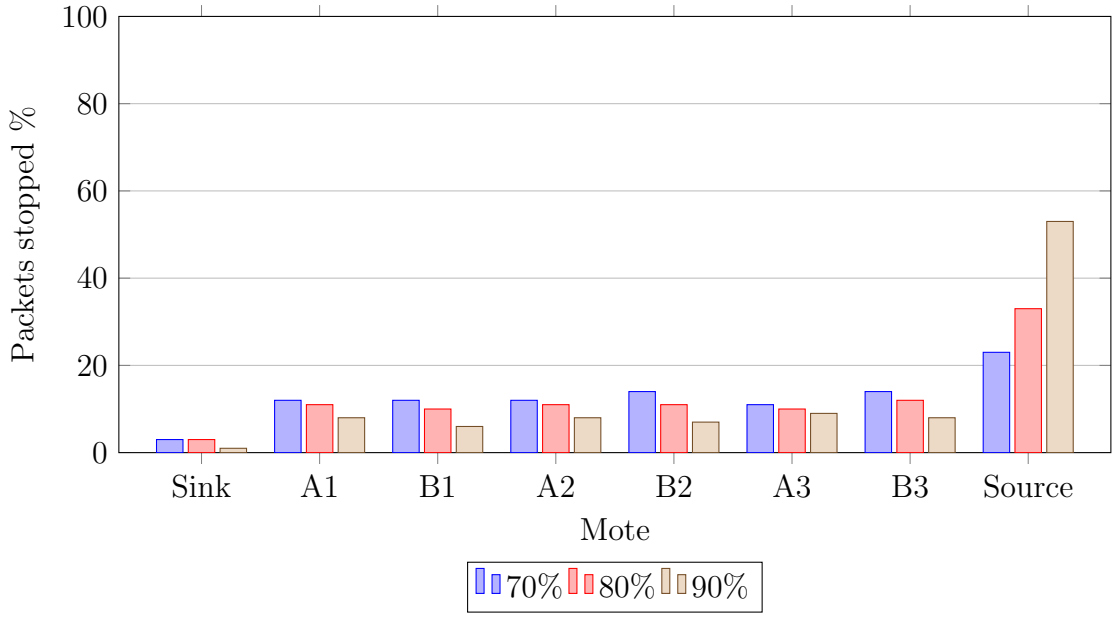


Figure 6.6: Packet elimination distribution for $\tau = 8$ with 70-90% link quality

Next, with a link quality of 80% Figure 6.7 show how $\tau = 816$ and $\tau = 1624$ shifts the packet elimination distribution. It shows longer delay enable RPE packets to return to source and stop packet being transmitted at Path B. $\tau = 816$ and $\tau = 1624$ yielded similar results. However, with $\tau = 816$ a few packets was dropped along the paths, but over 30 runs the average value recorded was less than 0.1% . Although, at B3 an average of 4 packet was eliminated yielding 0.2% of the

drops. Summarizing, $\tau = 816$ and $\tau = 1624$ behaves similarly with 99.76% and 100% of eliminations at sink. Moreover, further effects of the delays are discussed later on.

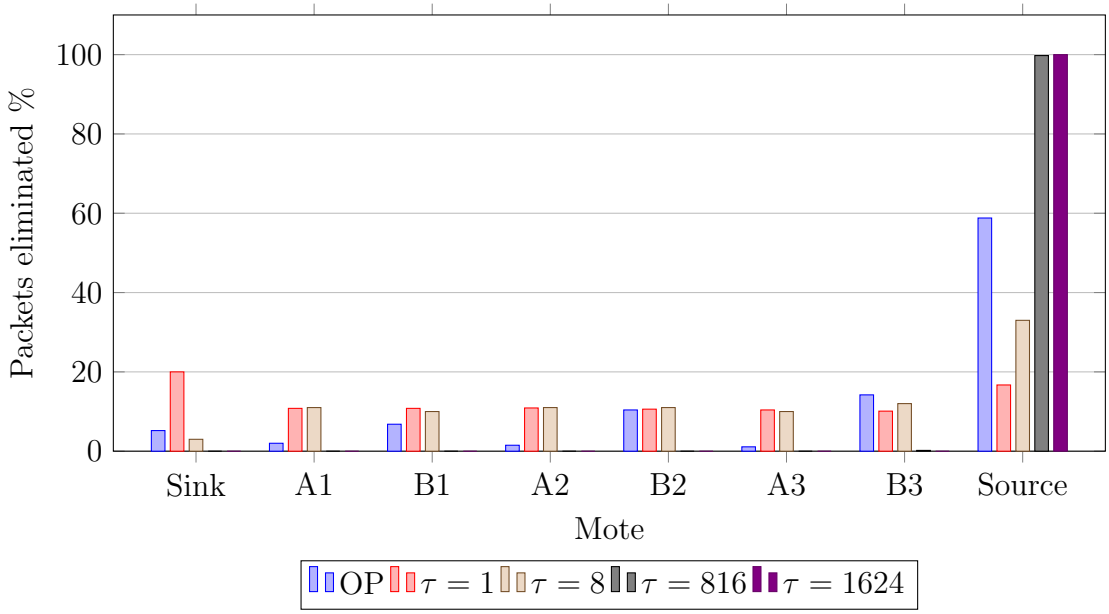
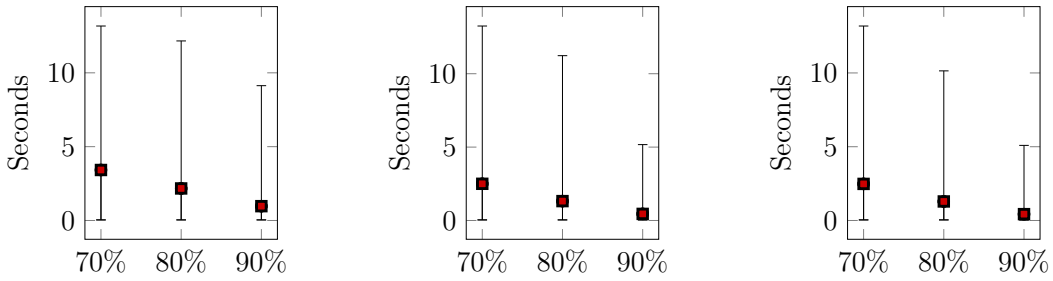


Figure 6.7: Packet elimination distribution with 80% link quality

6.4 Latency

This section present simulated latencies in terms of average, minimum, maximum, 99th percentile and variation in delay.



(a) Single Path (b) $\tau = 8$ (c) $\tau = 1$

Figure 6.8: Latencies for Single Path, $\tau = 8$ and $\tau = 1$. Top line shows maximum, box indicates average and bottom line is minimum latency

6.4.1 Average latency

In Figure 6.8(a) latencies for Single Path are visualized. There is decline in average latency with regards to a higher PDR. Average latency is higher compared to Figure 6.8(b) and Figure 6.8 (c) which show latencies for $\tau = 8$ and $\tau = 1$. Results are listed in Table 6.9. Next, given 70% link quality latency is lowered 94 milliseconds with $\tau = 1$ and 92 milliseconds with $\tau = 8$ compared to Single Path. At 90% effects of packet replication are clear with Single Path having a 125.5% higher latency than $\tau = 1$ and 115.5% higher than $\tau = 8$.

PDR	**Single Path** [sec]	$\tau = 1$ [sec]	$\tau = 8$ [sec]
70%	3.42	2.48	2.5
80%	2.17	1.29	1.32
90%	0.97	0.43	0.45

Table 6.9: Average latencies with different link qualities in seconds

6.4.2 Minimum latency

All simulated scenarios achieved minimum latencies as calculated in Subsection 6.1.1. Hence, 40 milliseconds is lowest latency for all scenarios. On the other hand, when overprovisioning a minimum latency of 70 milliseconds was recorded. This is as mentioned in Subsection 6.1.1 previously due to extra cells in tracks.

6.4.3 Maximum latency

Maximum registered latencies are not averaged over 30 runs, but the highest value recorded for all runs are presented. What is more, maximum latencies are within theoretical maximums as shown in Table 6.10. Taking a closer look, Single Path with 13.16 seconds is closest with 3 seconds from reaching its theoretical maximum delay.

PDR	**Single Path** [sec]	$\tau = 1$ [sec]	$\tau = 8$ [sec]
70%	13.16	13.18	13.25
80%	12.16	10.14	11.23
90%	9.13	5.09	5.17

Table 6.10: Max latencies with different link values in seconds

6.4.4 99th percentile

How latencies distributes in the 99th percentile are presented in Table 6.11. Single Path has a higher 99th percentile for all links as compared to $\tau = 1$ and $\tau = 8$. At 90% difference is 1.89 seconds between $\tau = 1$ and Single Path. Increasing τ from 1 to 8 increases 99th percentile latency with 40 milliseconds. This interesting as the difference in sending delay twice as high with 80 milliseconds.

PDR	**Single Path** [sec]	$\tau = 1$ [sec]	$\tau = 8$ [sec]
70%	9.21	7.49	7.53
80%	7.08	4.89	4.96
90%	4.08	2.19	2.23

Table 6.11: 99th percentile latencies with different link values in seconds

Decreasing link quality increases 99th percentile latencies for all scenarios. What is more, difference between Single Path $\tau = 1$ and $\tau = 8$ are fairly identical with 70%, 80% and 90% link quality. Another key point is delivery ratio. With packet replication reliability is increased as compared to Single Path as shown in Section 6.2. but, it is important to note that with packet replication more packet are received at sink. With Single Path, these packets are lost and hence not shown in recorded latencies.

6.4.5 All scenarios

Summarizing results for 80% link quality and adding longer delays $\tau = 816$ and $\tau = 1624$ are presented in Figure 6.9 and Table 6.12.

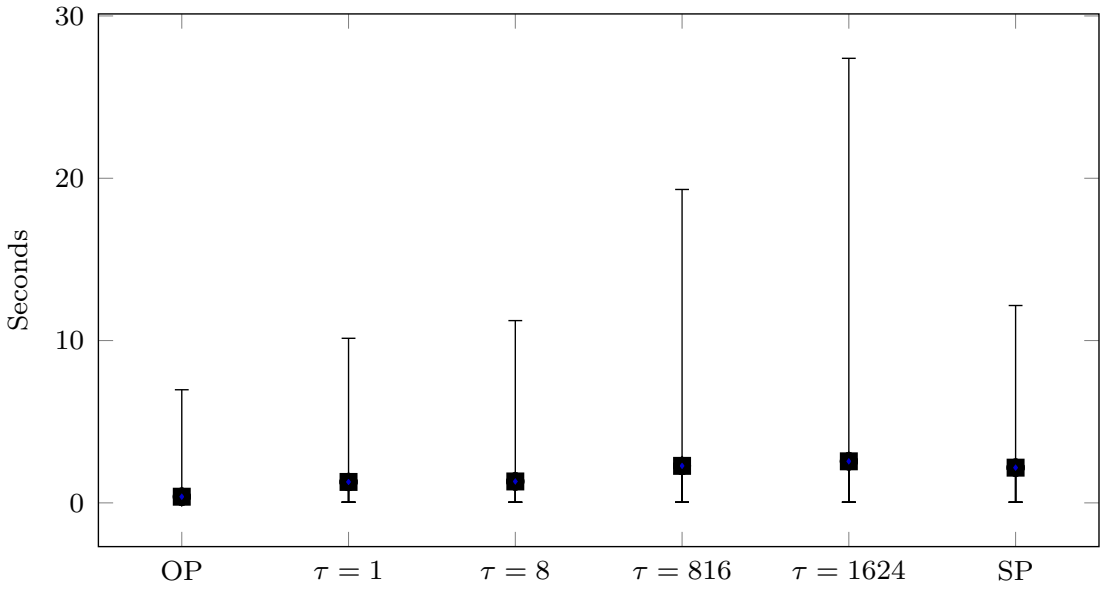


Figure 6.9: Latencies for all versions in seconds with 80% link quality. Top line shows maximum, box indicates average and bottom line is minimum latency

Average latency do not increase significantly from Single Path when longer delays $\tau = 816$ and $\tau = 1624$ are introduced. For example, $\tau = 816$ average is only 11 milliseconds higher than Single Path average. But, max and 99th percentile latencies increases significantly with increased τ. This is expected due to decrease in packet loss as shown in Table 6.8. In essence, packets with longer replication delays are received which increases recorded latencies.

Method/Delay	**Average latency** [sec]	**99%** [sec]	**Max latency** [sec]
Single Path	2.17	7.09	12.16
$\tau = 1$	1.29	4.90	10.14
OP	0.38	2.46	6.97
$\tau = 8$	1.32	4.96	11.23
$\tau = 816$	2.28	10.34	19.31
$\tau = 1624$	2.56	18.43	27.39

Table 6.12: Average, 99th percentile and maximum latencies for all scenarios with 80 % link quality

RPE Overprovisioning

Overprovisioning was introduced with 80% link quality, and as listed in Table 6.12. Average latency for Single Path is 2.17 seconds compared to 0.38 seconds when overprovisioning 1 extra TX and RX cell. This show Single Path has a 417% higher average latency than a overprovisioned path.

Variation in delay

Taking a closer look at variation between maximum and minimum latencies with a link quality of 80% reveals overprovisioning has smallest difference, Δ, with 6.9 seconds between highest and lowest latency. On the other hand, $\tau = 1624$ records highest difference with 27.35 seconds. All differences, Δ, are presented in Table 6.13.

Method/Delay	**Minimum latency** [sec]	**Max latency** [sec]	Δ [sec]
Single Path	0.04	12.16	12.12
$\tau = 1$	0.04	10.14	10.10
$\tau = 8$	0.04	11.23	11.19
$\tau = 816$	0.04	19.31	19.27
$\tau = 1624$	0.04	27.39	27.35
OP: $\tau = 8$	0.07	6.97	6.90

Table 6.13: Difference Δ between maximum and minimum latency in seconds with 80% link quality

These results are expected as theoretical maximum latencies increase with longer replication delay τ as shown in Subsection 6.1.1. In short, decreasing replication delay τ reduces variation in delay.

6.5 Mote lifetime

This section present simulated lifetimes in terms of average mote life, average network life and lowest mote life. Mote lifetime is important, as it shows effects of increasing reliability and decreasing latencies, two important factors in industrial WSNs. Moreover, as mentioned earlier in Subsection 5.9.4 parent selection influences DAO messages, and in return battery life. This is due to the fact that in 9 of 10 slotframe iterations source has Path B as is route to sink. It can be seen in Dual Path where A3 and B3 behaves similarly, but lifetime is approximately 0.5 years lower at Path B.

6.5.1 Average mote life

In Figure 6.10 average mode lifetimes for all packet replication versions are presented with averaged values. Due to extra TX/RX cell when Overprovisioning, Path A has a significantly lower battery life as compared to Path B and all other versions. Next, at all motes, expect when Overprovisioning, Dual Path has the lowest battery life. Looking at lifetimes for all versions, there is a clear similarity with where the upstream copy was eliminated. $\tau = 816$, $\tau = 1624$ and Overprovisioning has highest number of packets eliminated at source as seen in Section 6.3. Hence, highest battery life at source as it do not send two copies as often as other scenarios. This is supported by the theoretical lifetime, where motes sending RPE packets have a higher excepted mote lifetime. Interestingly, $\tau = 1$ decrease load on A1 due to highest drop rate at sink, A1 and B1. This effect is discussed in Section 6.6. On the other hand, it causes the source to have lowest battery life of all RPE scenarios.

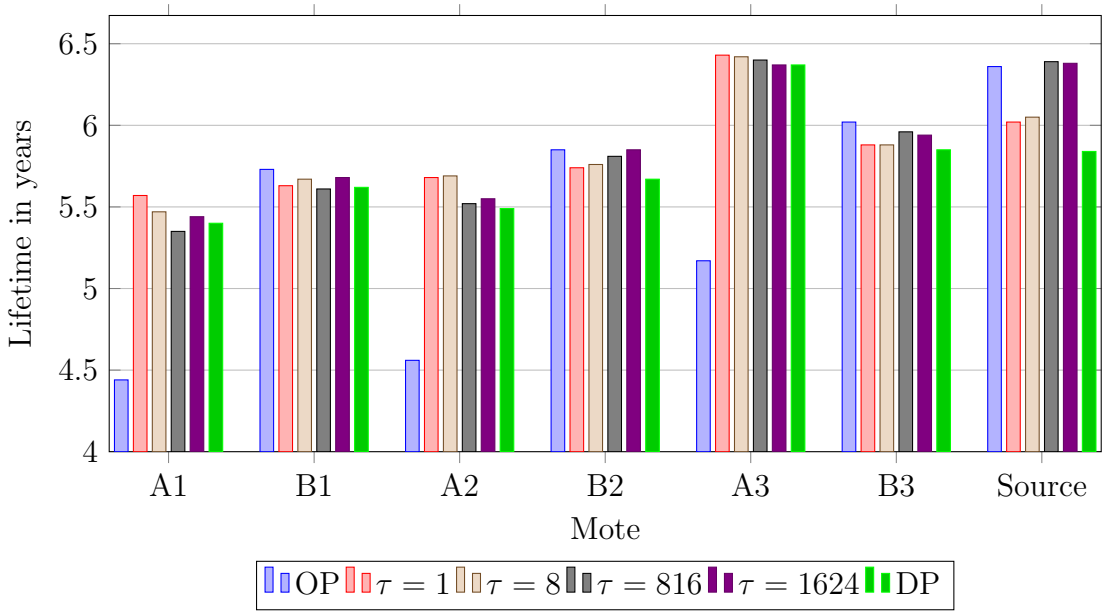


Figure 6.10: Scenarios and their average mote lifetimes in years

6.5.2 Lowest mote life

Lowest mote life represents lowest average mote lifetime in all simulations. For all simulations, first mote to run out of power is A1, except in $\tau = 1$ and $\tau = 8$ where in approximately $\frac{1}{30}$ runs B1 ran out first. Lowest mote lifetimes are depicted in Figure 6.11. Looking at Path A, no elimination was registered when using $\tau = 816$ or $\tau = 1624$ causing traffic to behave as DP and SP. This is reflected in

lowest mote lifetime as all of them run out of power within 10 days of each other. Moreover, with $\tau = 1$ and $\tau = 8$ lowest lifetime is increased with minimum 33 days compared to Single Path, with $\tau = 8$ prevailing with 34 days extra life. Effects of Overprovisioning are clear with A1 running out of battery 338 days before any other version.

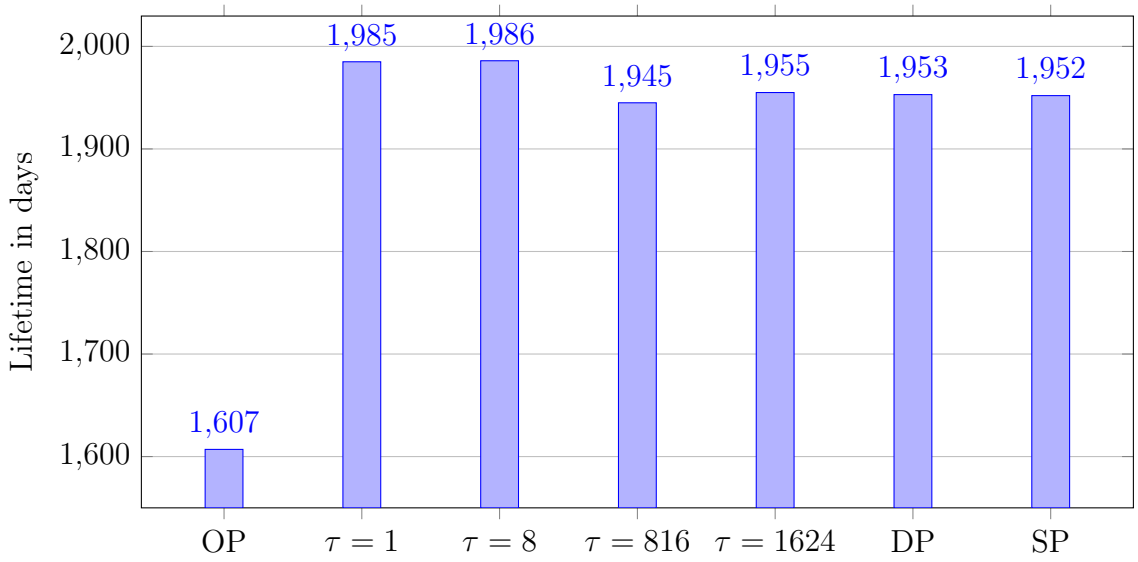


Figure 6.11: Scenarios and their lowest mote life registered in days

6.5.3 Average current consumption

Another way of looking into effects of the different scenarios are average current consumption. In fact, effects of packet replication are clear in terms of lifetime as Single Path utilizes 159 mA less than second place $\tau = 1624$. Average current consumption is shown in Figure 6.12

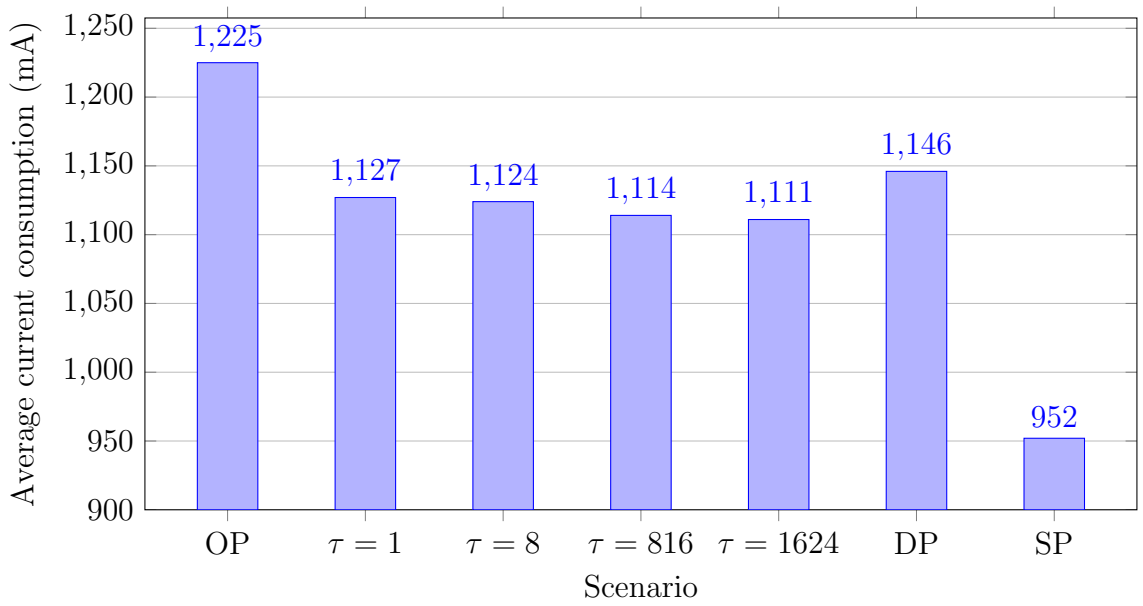


Figure 6.12: Average current consumption in the network in mA

Effects of overprovisioning are seen with the highest current consumption at 1225 mA. This increase is due to the extra cells, which causes more idle listening than other scenarios. Another key point, looking at RPE versions compared to Dual Path there is a decline in average current consumption. A decrease in average current consumption is important as this topology no not represent a network as a whole. In fact, with a real network data traverse from other paths as well. Hence, a decreased current consumption is desired to decrease load on certain motes due to funneling effects. This will be discussed in Section 6.6.

6.5.4 Comparing RPE with Single Path

Taking a closer look at RPE with $\tau = 8$ and Single Path with link quality of 70% reveal at Path A, RPE outperform Single Path with higher lifetimes as illustrated in Figure 6.13. This is due to RPE packets dropping upstream packet stuck in retransmission. On the other hand, at Path B, Single Path has significantly higher lifetime as no traffic is flowing through it. At source, Single Path has a higher life as it send one packet compared to RPE with two packets. However, in terms of reliability the network may be considered down when the first mote is out of power, and at 70% link quality RPE has 29 days longer life, a 1.53% increase.

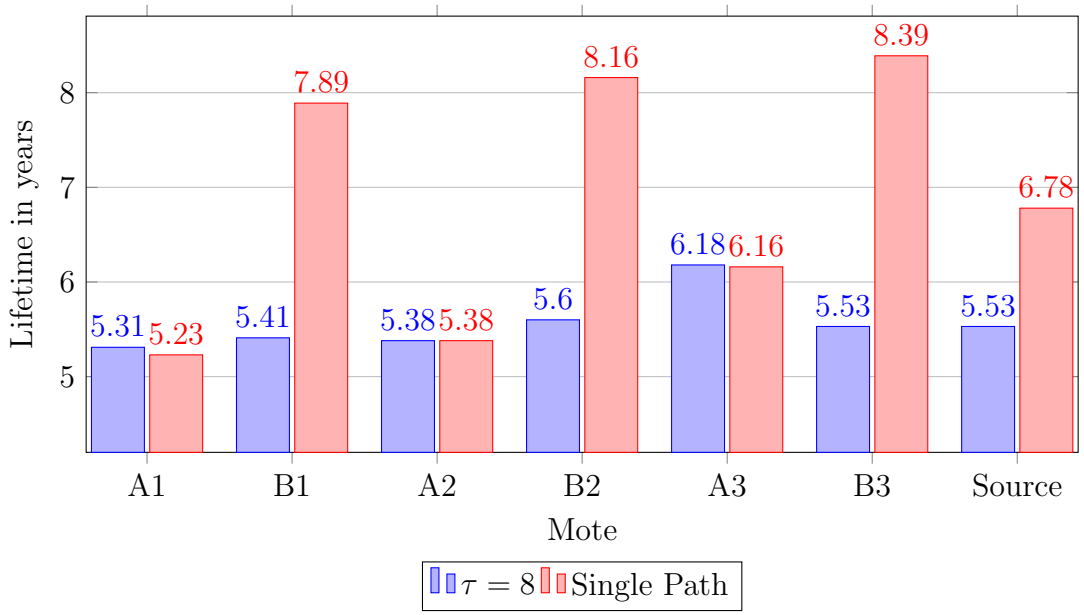


Figure 6.13: Lifetimes for Single Path and RPE $\tau = 8$ with 70% link quality

6.6 Funneling effect

As presented in Subsection 2.10, motes closer to sink have a higher load than motes further away. When utilizing RPE with shorter replication delays τ, it was observed that the funneling effect was decreased compared to the other scenarios. This is shown in Figure 6.14, where average lifetime at A1 and B1 for all versions using packet replication are listed. Having a short delay $\tau = 1$ return the most even distribution between A1 and B1 due to the amount of packet elimination happening at these motes. In addition, $\tau = 8$ decreases funneling effect as well, but as much as $\tau = 1$. On the other hand, increasing replication delay τ decreases load on Path B. Moreover, Overprovisioning has lowest latency, resulting in highest load on Path A, and lowest load on Path B. Funneling effect can be seen for all motes in Figure 6.10 where lifetime increases with distance from sink.

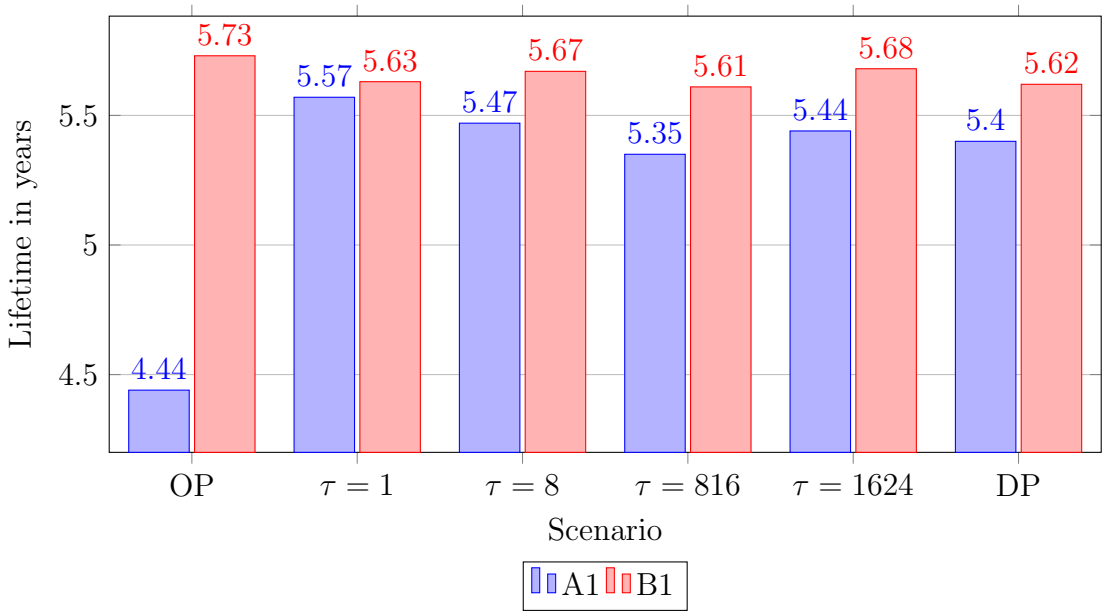


Figure 6.14: Different scenarios and their funneling effect at A1 and B1. Represented with mote lifetime in years

Chapter 7

Discussion

This chapter discuss the results and present advantages and disadvantages of introducing different delay τ in packet replication. Ideally, results should be compared with related work Leapfrog Collaboration, but they have utilized Contiki (briefly presented in Section 4.1) and as mentioned in Subsection 1.5.1 their implementation was not available. LFC compares results to their own implementation without LFC. Hence, results of RPE are compared to Single Path and Dual Path. As illustrated in Chapter 6, there are a lot factors to take into consideration, and getting an overview of all is a challenge. Such as latency, reliability and average current consumption. Therefor, in this section all traits are plotted in radar charts to visualize the attributes. But, to understand the visualization a short description of the chart is provided.

7.1 Plot description

All charts are presented with values in percent. The radar charts have lattice with 10 lines, each representing increments of 10%. Outer line is 100% and represent the worst recorded value for that trait. For instance, for 80% link the highest maximum latency recorded is Single Path with 12.16 seconds. For $\tau = 8$ the same trait is 11.23 seconds, a 7.6% decrease. In the chart these would show up as 100% and 92.3%. To summarize, a plot closer to centre indicates a better result.

7.2 Discussion of Results

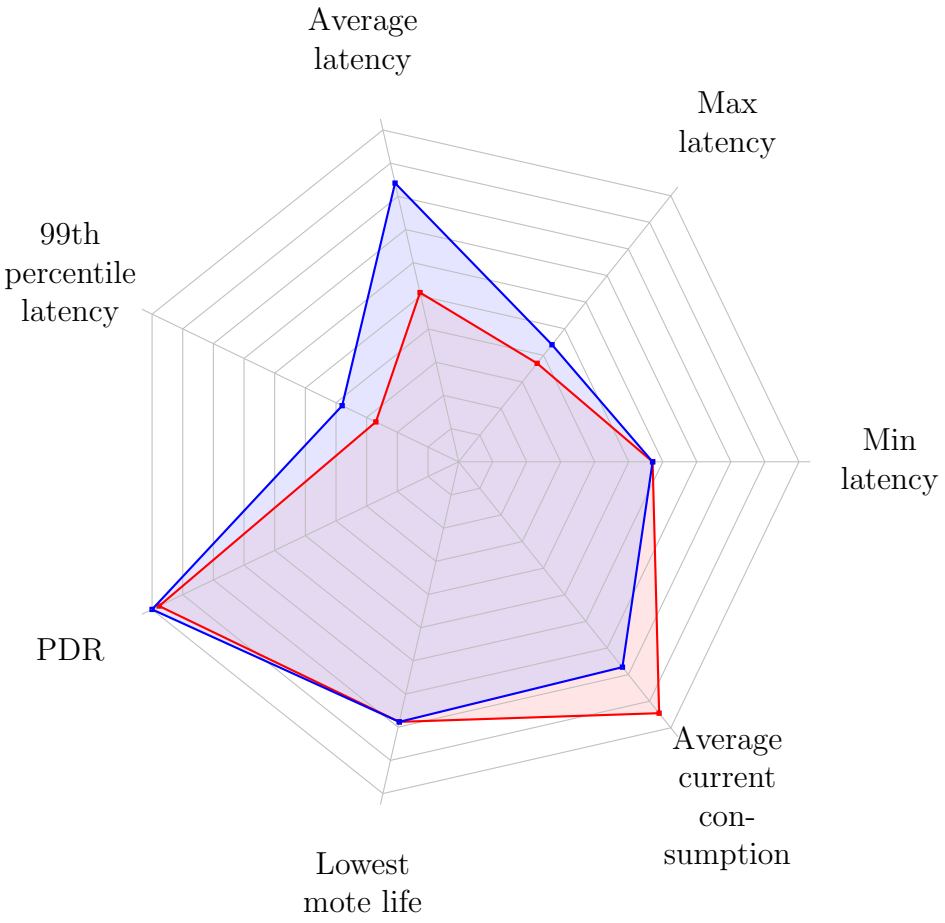


Figure 7.1: Representation of Dual Path (red) and Single Path (blue) traits with 80 link quality

Firstly, lets start by discussing results with a link quality of 80%. Figure 7.1 shows effects of utilizing packet replication by comparing Single Path and Dual Path. In fact, Single Path has lower average current consumption, but in terms of latency and PDR it worsens. Moreover, with packet replication, latencies decreases significantly as shown with Dual Path. What is more, PDR is increased with 2.25% compared to Single Path.

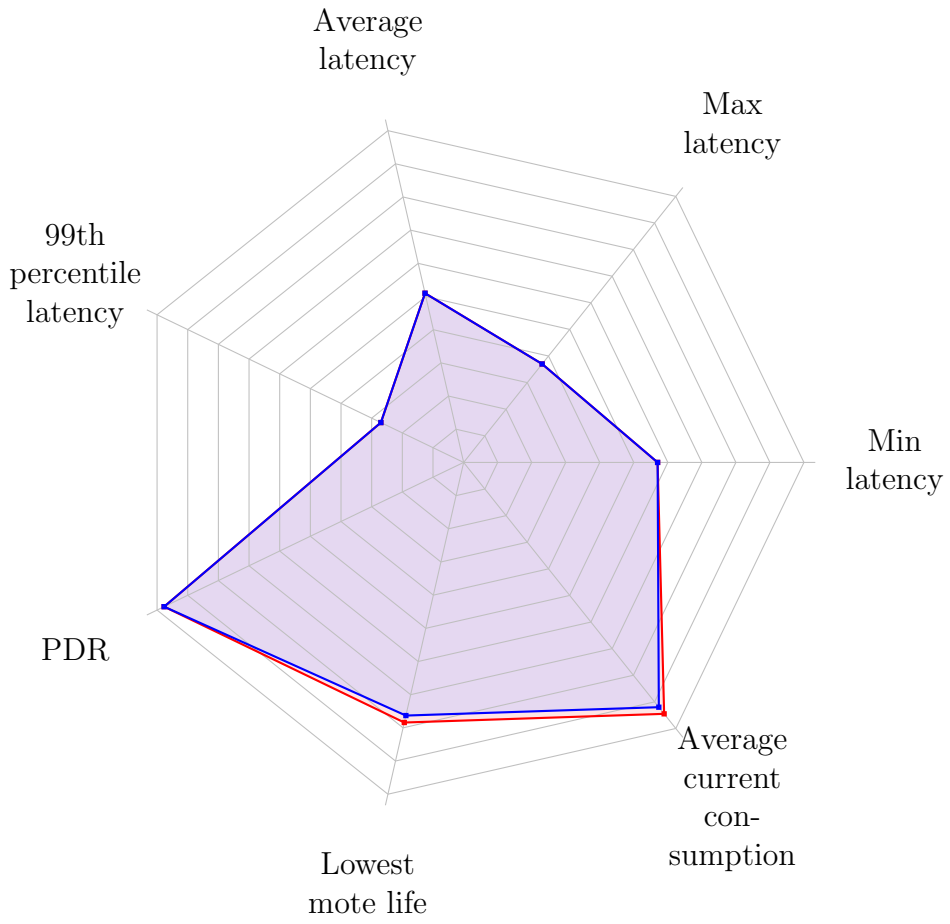


Figure 7.2: Representation of $\tau = 1$ traits (blue) compared to Dual Path (red) with 80% link quality

Next, effects of $\tau = 1$ do not change anything in terms of latency with regards to Dual Path, as they behave similarly upstream. But, due to downstream elimination average energy consumption is decreased, and lowest battery life is increased as depicted in Figure 7.2.

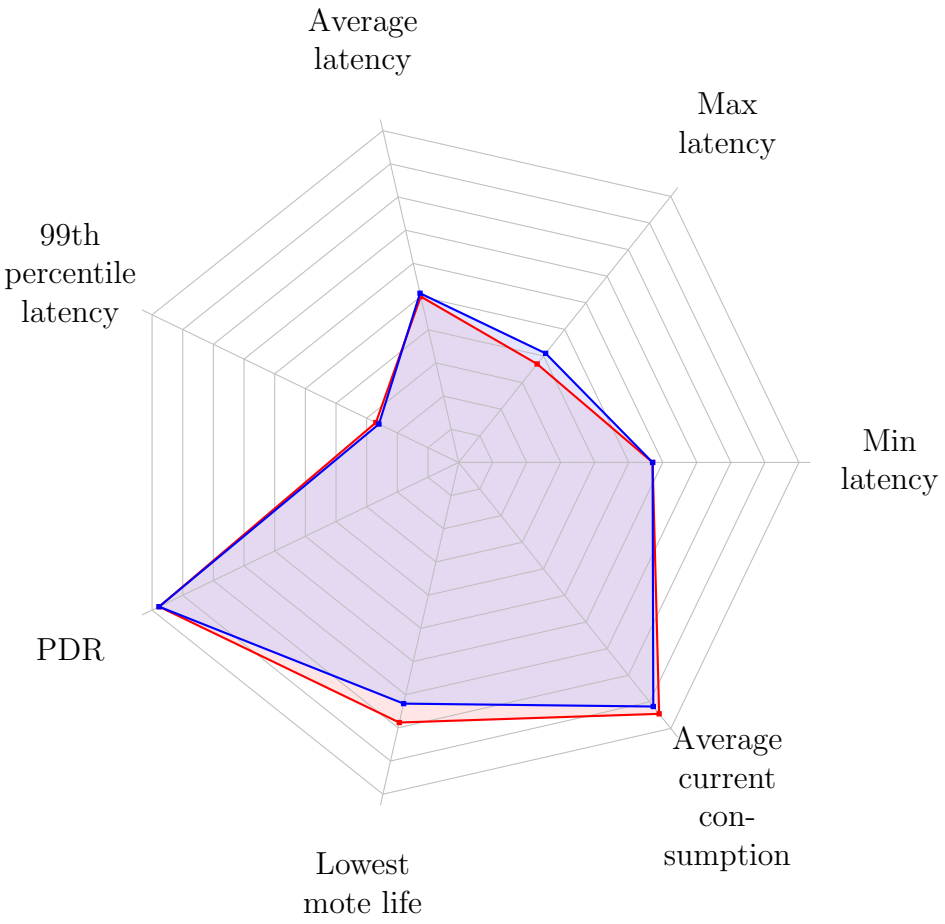


Figure 7.3: Representation of $\tau = 8$ traits (blue) compared to Dual Path (red) with 80% link quality

When introducing $\tau = 8$, latencies are increased slightly compared to Dual Path. Max latency with 4% increase, average latency with 1% and 99th percentile with 1%. On the other hand, this extra delay increases lowest mote life and decreases average current consumption as shown in Figure 7.3.

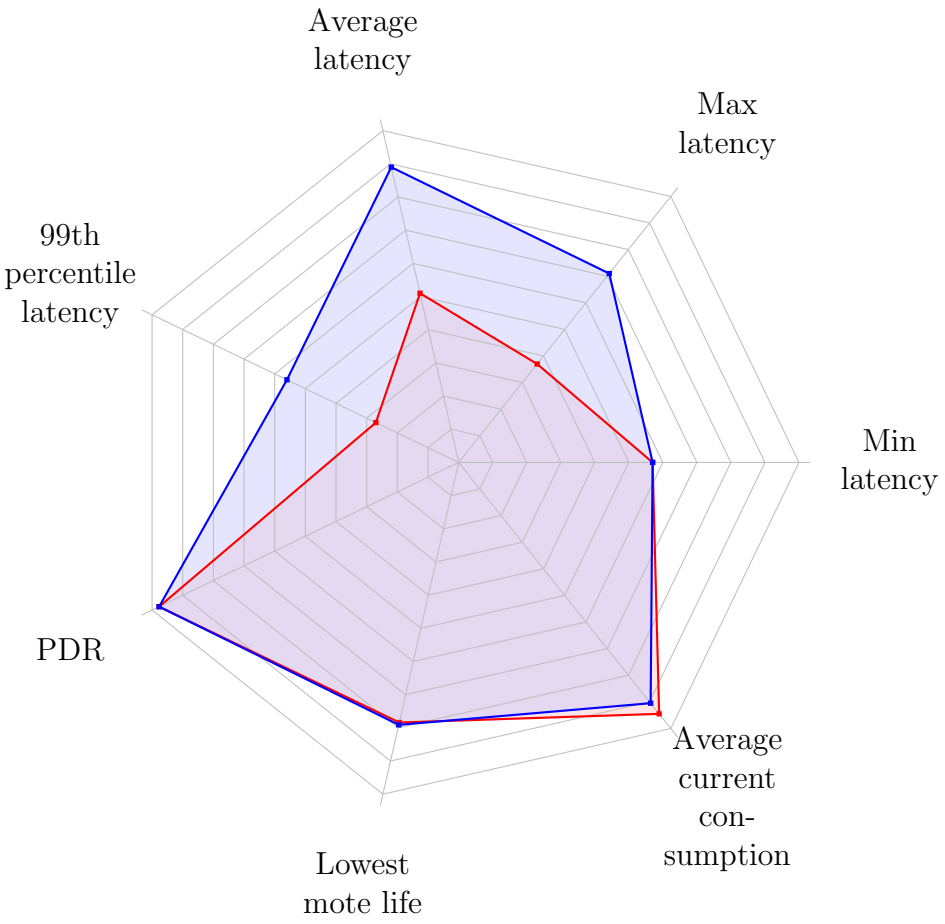


Figure 7.4: Representation of $\tau = 816$ traits (blue) compared to Dual Path (red) with 80% link quality

Extending delay further with $\tau = 816$ increases effects shown with $\tau = 8$. Figure 7.4 show this with $\tau = 816$ where 99th percentile, average and maximum latency are increased significantly. However, average current consumption decreases.

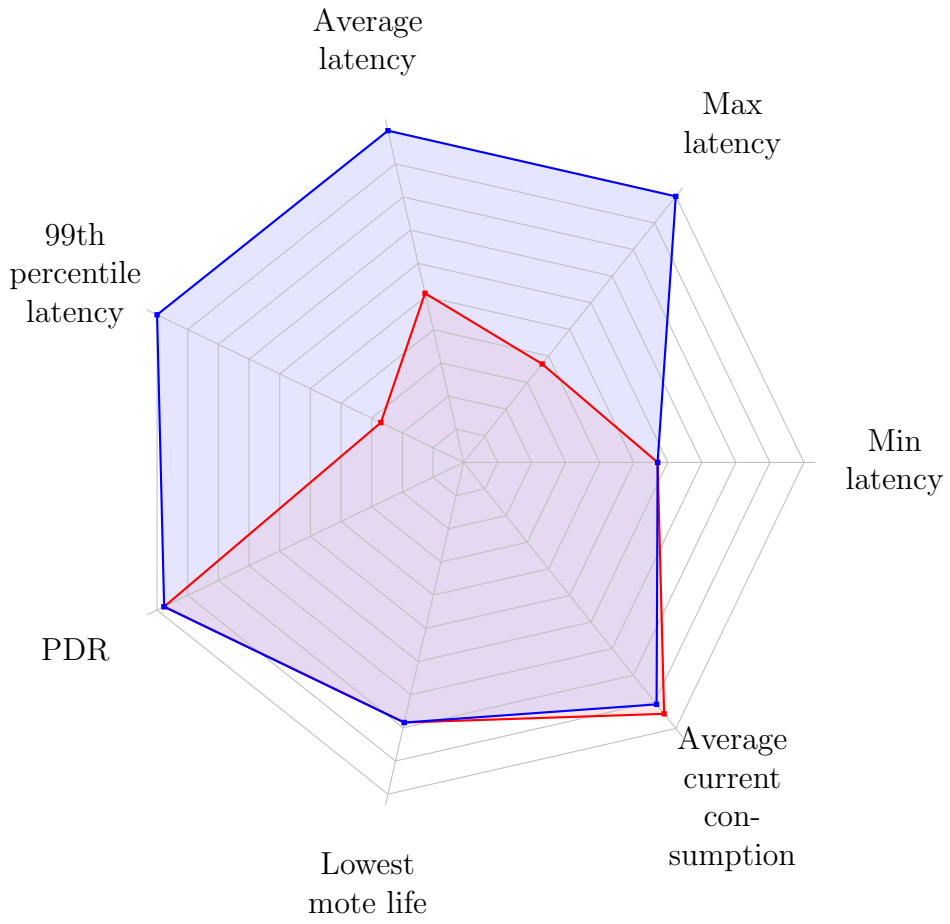


Figure 7.5: Representation of $\tau = 1624$ traits (blue) compared to Dual Path (red) with 80% link quality

Doubling delay to $\tau = 1624$ decreases average current consumption of the network, but with this long replication delay, latencies are the highest recorded. In terms of lowest mote life, $\tau = 1624$ has similar results as Dual Path depicted in Figure 7.5.

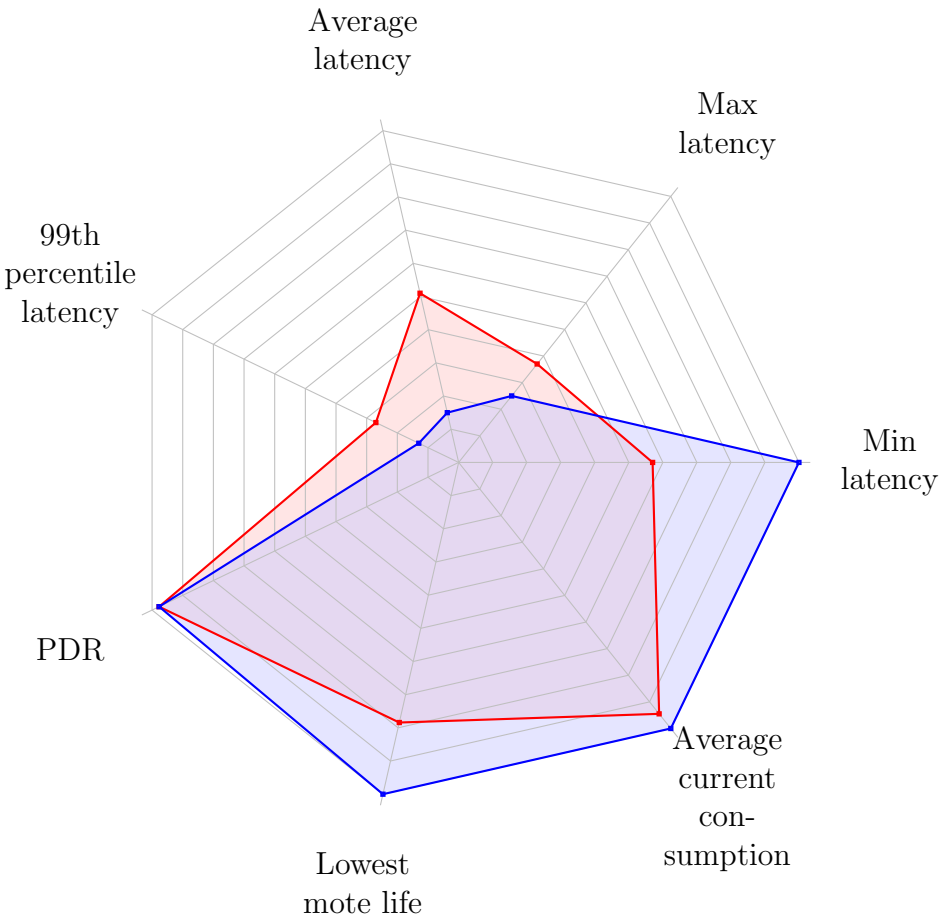


Figure 7.6: Representation of overprovisioning traits (blue) compared to Dual Path (red) with 80% link quality

When overprovisioning, results become clear as shown in Figure 7.6. It results in highest average current consumption and lowest mote life. But, latencies are reduced drastically with the exception that minimum latency is increased due to the extra TX and RX cells.

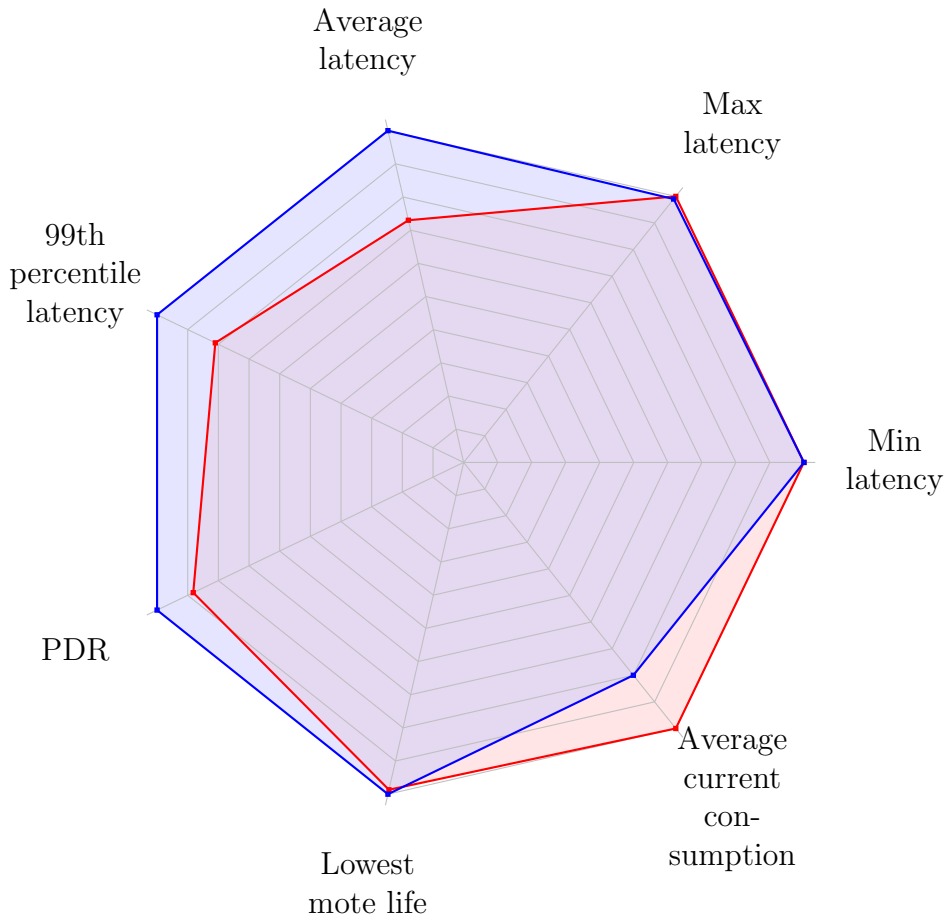


Figure 7.7: Single Path (blue) compared to $\tau = 8$ (red) traits with 70% link quality

Effects of changing link quality are presented in Figure 7.7. In this Figure, links are set to 70% and Single Path is plotted against $\tau = 8$. In this example, advantages of utilizing RPE become apparent. Firstly, PDR is increased with 10.5% from 88.2% to 98.7%. In addition, a 27% reduction in average latency and 19% decrease in terms of 99th percentile latency. Maximum and minimum latencies are somewhat identical. Interestingly, by utilizing RPE network life is increased with 1.5%. But, average current consumption is in return increased with 19.8%.

7.2.1 Summarizing discussions

Results shows $\tau = 8$ is optimal delay. Since in a perfect environment the tracks allows a packet to loop back and stop source from sending the replicated packet. Based on the results with 80% link quality, $\tau = 8$ achieves best traits of all replication delays. In fact, as compared to Dual Path and Single Path lowest mote lifetime is increased with 1.74%. Next, average latency is reduced with 39.1% and reliability increases from 97.7% to 99.95% from Single Path. A natural disadvantage with the RPE proposal is the increase in average current consumption. This is due to extra tracks and packets traversing disjoint paths. However, packet replication is recommended by DetNet architecture and RPE do in fact decreases average current consumption with regards to Dual Path. But, it increases with 18% compared to Single Path.

Finally, if a low average latency is critical, it is recommended to utilize Overprovisioning. Overprovisioning has a 71.2% lower average latency than $\tau = 8$. However, this comes with a 18.34% reduction in lowest mote lifetime.

Chapter 8

Conclusion

This thesis studied effects of a Reverse Packet Elimination (RPE) algorithm in 6TiSCH. Specifically, effects of delaying packets when utilizing packet replication. In short, advantages are higher reliability, lower average latency and increased mote lifetime. However, average energy consumption in the network increases.

Related work Leapfrog Collaboration (LFC) delivers a 99.1% worst case reliability, with a disadvantage of increasing energy consumption. On the other hand, RPE decreases energy consumption compared to not utilizing a reverse elimination technique, and delivers a worst case reliability of 97.7%. Another key point, lowest mote life increases as RPE decreases funneling effect of motes closer to sink. Moreover, compared to a single point of failure, latency with RPE decreases. A drawback of RPE versus LFC is a slight increase in average latency due to the delay in the replication mechanism. LFC delivers a 41% reduction in average latency compared to standard retransmission schemes, while RPE delivers a 39% reduction. Moreover, LFC is by design dependent on high node density, but RPE only require two disjoint paths towards sink. This makes RPE less dependable on topology and presumably suitable for other WSNs.

All in all, as recommended by DetNet architecture and to be adopted in 6TiSCH, packet elimination and replication is vital to achieve high end-to-end reliability. As revealed by this thesis, RPE achieves promising results, but should be further investigated and studied if to be deployed in a real network.

8.1 Future work

This section discuss ideas not implemented but thought of during creation of RPE. These are suggestions, and have not been tested nor verified.

Number of paths Currently, RPE is limited to two disjoint paths, but extra paths could be implemented. This would increase reliability further with a drawback of increased energy consumption. However, it would require more motes within sensing range.

Preferred path Next, in all simulations Path A has been scheduled first. Furthermore, link quality have been static, but in the real world link qualities are dynamic. Implementing a counter at sink allows it to keep track of which path arrives first can utilized to switch path priorities. For instance, if Path A packet is sent before Path B packet, but Path B packet arrives at sink first indicate there is a bad link at Path A. A maximum constraint could be set, and if the counter surpasses it, sink sends a packet down Path B to source and tell it to switch priority. Theoretically, this would lower average latency. Another option, is that after a predefined number of slotframe iterations, sink sends a packet down both paths simultaneously. Then, the path arriving first at source would be set as preferred path.

Decreasing idle slots In the current PCE design, it schedules 1 TX cell each slotframe. However, the TX cell is only utilized every 10th slotframe. A centralized controller could ramp up and ramp down slots not utilized. For instance, all track slots after the 4th is by design never receiving or transmitting as the number of attempts are maxed out. In short, 50% of slots in a tracks are currently never utilized. In essence, idle slots could be put in sleep mode with careful design.

Load balancing As mentioned previously, Path A packet is scheduled first, and in the paragraph about Preferred path the idea of switching paths in the event of a link quality deteriorating is discussed. However, another aspect is to alternate between paths by default. This can presumable distribute the load at both paths as an RPE packet utilizes less resources.

Decreasing retransmission attempts Assuming link qualities do not deteriorate, decreasing retransmission attempts from 4 to 2 when utilizing packet replication with disjoint paths would theoretically have as many attempts as Single Path. Hence, RPE could in theory achieve similar end-to-end reliability as Single Path with a decrease in average current consumption.

Security considerations With packet replication there are multiple paths for a man-in-the-middle attack. These attacks are assessed by DetNet in their security consideration [48]. But, with the introduction of RPE another aspect needs to be considered. Theoretically, an attacker could spoof downstream RPE packets and send them towards the source. This would cause source or other motes to never forward packets upstream as they are told to drop them. In essence, a Denial of Service attack were data packets never reach the sink.

Bibliography

[1] J. T. Adams, "An introduction to ieee std 802.15.4," in *2006 IEEE Aerospace Conference*, pp. 8 pp.–, March 2006.

[2] X. Vilajosana, Q. Wang, F. Chraim, T. Watteyne, T. Chang, and K. S. J. Pister, "A realistic energy consumption model for tsch networks," *IEEE Sensors Journal*, vol. 14, pp. 482–489, Feb 2014.

[3] E. Municio, G. Daneels, M. Vučinić, S. Latré, J. Famaey, Y. Tanaka, K. Brun, K. Muraoka, X. Vilajosana, and T. Watteyne, "Simulating 6tisch networks," *Transactions on Emerging Telecommunications Technologies*, vol. 30, no. 3, p. e3494, 2019. e3494 ett.3494.

[4] "Feedback control system or closed loop control system, howpublished = `http://instrumentationandcontrollers.blogspot.com/2011/05/feedback-control-system-or-closed-loop.html`, note = Accessed: 2019-04-19."

[5] J. d. Armas, P. Tuset, T. Chang, F. Adelantado, T. Watteyne, and X. Vilajosana, "Determinism through path diversity: Why packet replication makes sense," in *2016 International Conference on Intelligent Networking and Collaborative Systems (INCoS)*, pp. 150–154, Sep. 2016.

[6] R. Koutsiamanis, G. Z. Papadopoulos, X. Fafoutis, J. M. D. Fiore, P. Thubert, and N. Montavont, "From best effort to deterministic packet delivery for wireless industrial iot networks," *IEEE Transactions on Industrial Informatics*, vol. 14, pp. 4468–4480, Oct 2018.

[7] G. G. Lorente, B. Lemmens, M. Carlier, A. Braeken, and K. Steenhaut, "Bmrf: Bidirectional multicast rpl forwarding," *Ad Hoc Networks*, vol. 54, pp. 69 – 84, 2017.

[8] G.-S. Ahn, S. G. Hong, E. Miluzzo, A. T. Campbell, and F. Cuomo, "Funneling-mac: A localized, sink-oriented mac for boosting fidelity in sensor

networks," in *SenSys'06: Proceedings of the Fourth International Conference on Embedded Networked Sensor Systems*, pp. 293–306, 10 2006.

[9] IEEE, "Ieee standard for low-rate wireless networks," *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pp. 1–709, April 2016.

[10] K. Wehrle, M. Günes, and J. Gross, *Modeling and Tools for Network Simulation*. Springer, 01 2010.

[11] V. C. Gungor and G. P. Hancke, "Industrial wireless sensor networks: Challenges, design principles, and technical approaches," *IEEE Transactions on Industrial Electronics*, vol. 56, pp. 4258–4265, Oct 2009.

[12] D. Dujovne, T. Watteyne, X. Vilajosana, and P. Thubert, "6tisch: deterministic ip-enabled industrial internet (of things)," *IEEE Communications Magazine*, vol. 52, pp. 36–41, December 2014.

[13] A. N. Kim, F. Hekland, S. Petersen, and P. Doyle, "When hart goes wireless: Understanding and implementing the wirelesshart standard," in *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, pp. 899–907, Sep. 2008.

[14] F. P. Rezha and S. Y. Shin, "Performance evaluation of isa100.11a industrial wireless network," in *IET International Conference on Information and Communications Technologies (IETICT 2013)*, pp. 587–592, April 2013.

[15] L. Doherty, W. Lindsay, and J. Simon, "Channel-specific wireless sensor network path data," in *2007 16th International Conference on Computer Communications and Networks*, pp. 89–94, Aug 2007.

[16] P. Thubert, "An Architecture for IPv6 over the TSCH mode of IEEE 802.15.4," Internet-Draft draft-ietf-6tisch-architecture-20, Internet Engineering Task Force, Mar. 2019. Work in Progress.

[17] N. Finn, P. Thubert, B. Varga, and J. Farkas, "Deterministic Networking Architecture," Internet-Draft draft-ietf-detnet-architecture-10, Internet Engineering Task Force, Dec. 2018. Work in Progress.

[18] P. Park, S. Coleri Ergen, C. Fischione, C. Lu, and K. H. Johansson, "Wireless network design for control systems: A survey," *IEEE Communications Surveys Tutorials*, vol. 20, pp. 978–1013, Secondquarter 2018.

[19] J. Araujo, J. Lázaro, A. Astarloa, A. Zuloaga, and A. Garcia, "Prp and hsr version 1 (iec 62439-3 ed.2), improvements and a prototype implementation," pp. 4410–4415, 11 2013.

[20] G. Z. Papadopoulos, T. Matsui, P. Thubert, G. Texier, T. Watteyne, and N. Montavont, "Leapfrog collaboration: Toward determinism and predictability in industrial-iot applications," in *2017 IEEE International Conference on Communications (ICC)*, pp. 1–6, May 2017.

[21] A. Nasrallah, A. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. El Bakoury, "Ultra-low latency (ull) networks: The ieee tsn and ietf detnet standards and related 5g ull research," *IEEE Communications Surveys & Tutorials*, 2018.

[22] B. Sklar, *Digital Communications: Fundamentals and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[23] T. Watteyne, M. R. Palattella, and L. A. Grieco, "Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement." RFC 7554, May 2015.

[24] "Deterministic Networking (detnet) , howpublished = `https://datatracker.ietf.org/wg/detnet/about/`, note = Accessed: 2018-12-19."

[25] "Time-Sensitive Networking (TSN) Task Group, howpublished = `https://1.ieee802.org/tsn/`, note = Accessed: 2018-05-19."

[26] D. O. Awduche, L. Berger, D.-H. Gan, T. Li, D. V. Srinivasan, and G. Swallow, "RSVP-TE: Extensions to RSVP for LSP Tunnels." RFC 3209, Dec. 2001.

[27] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing Architecture." RFC 8402, July 2018.

[28] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, "Software-Defined Networking (SDN): Layers and Architecture Terminology." RFC 7426, Jan. 2015.

[29] J. Farkas, N. Bragg, P. Unbehagen, G. Parsons, P. J. Ashwood-Smith, and C. Bowers, "IS-IS Path Control and Reservation." RFC 7813, June 2016.

[30] J. McManus, J. Malcolm, M. D. O'Dell, D. O. Awduche, and J. Agogbua, "Requirements for Traffic Engineering Over MPLS." RFC 2702, Sept. 1999.

[31] E. Grossman, "Deterministic Networking Use Cases," Internet-Draft draft-ietf-detnet-use-cases-20, Internet Engineering Task Force, Dec. 2018. Work in Progress.

[32] T. Watteyne, P. Tuset-Peiro, X. Vilajosana, S. Pollin, and B. Krishnamachari, "Teaching communication technologies and standards for the industrial iot? use 6tisch!," *IEEE Communications Magazine*, vol. 55, pp. 132–137, May 2017.

[33] Q. Wang and X. Vilajosana, "6tisch operation sublayer (6top) protocol (6p)," RFC 6TiSCH Operation Sublayer (6top) Protocol (6P), Internet Engineering Task Force, Jan. 2019.

[34] G. Montenegro, C. Schumacher, and N. Kushalnagar, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals." RFC 4919, Aug. 2007.

[35] R. Alexander, A. Brandt, J. Vasseur, J. Hui, K. Pister, P. Thubert, P. Levis, R. Struik, R. Kelsey, and T. Winter, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks." RFC 6550, Mar. 2012.

[36] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)." RFC 7252, June 2014.

[37] E. Nilsen, "Energy Consumption Investigation in WSN using OMNeT++," Master's thesis, University of Oslo, Norway, 2013.

[38] H. Karl and A. Willig, *Protocols and Architectures for Wireless Sensor Networks*. USA: John Wiley & Sons, Inc., 2005.

[39] X. Vilajosana, K. Pister, and T. Watteyne, "Minimal IPv6 over the TSCH Mode of IEEE 802.15.4e (6TiSCH) Configuration." RFC 8180, May 2017.

[40] "Omnet++." `https://omnetpp.org/`. Accessed: 2018-11-25.

[41] "ns-3 network simulator." `https://www.nsnam.org/`. Accessed: 2018-11-26.

[42] "Home cooja." `https://github.com/contiki-ng/contiki-ng/wiki`. Accessed: 2018-11-30.

[43] "Overview simulator." `https://bitbucket.org/6tisch/simulator/`. Accessed: 2018-11-30.

[44] K. P. Hanh-Phuc Le, Mervin John, "Energy-aware routing in wireless sensor networks with adaptive energy-slope control," *EE290Q-2 Spring 2009*, 2009.

[45] "Openmote platform, howpublished = `https://github.com/contiki-ng/contiki-ng/wiki/platform-openmote-cc2538`, note = Accessed: 2019-03-10."

[46] T. Chang, M. Vučinić, X. Vilajosana, S. Duquennoy, and D. Dujovne, "6TiSCH Minimal Scheduling Function (MSF)," Internet-Draft draft-ietf-6tisch-msf-03, Internet Engineering Task Force, Apr. 2019. Work in Progress.

[47] G. Papadopoulos, R.-A. Koutsiamanis, N. Montavont, and P. Thubert, "Exploiting Packet Replication and Elimination in Complex Tracks in LLNs," Internet-Draft draft-papadopoulos-paw-pre-reqs-01, Internet Engineering Task Force, Mar. 2019. Work in Progress.

[48] T. Mizrahi, E. Grossman, A. J. Hacker, S. Das, J. Dowdell, H. Austad, K. Stanton, and N. Finn, "Deterministic Networking (DetNet) Security Considerations," Internet-Draft draft-ietf-detnet-security-04, Internet Engineering Task Force, Mar. 2019. Work in Progress.

[49] Hirschmann, "Tsn - time sensitive networking." https://www.iiconsortium.org/about-us.htm, 2018. [Online; Accessed 20.04.18].

[50] S. Nsaibi, L. Leurs, and H. D. Schotten, "Formal and simulation-based timing analysis of industrial-ethernet sercos iii over tsn," in *Proceedings of the 21st International Symposium on Distributed Simulation and Real Time Applications*, pp. 83–90, IEEE Press, 2017.

[51] A. Alliance, "Our members." http://www.hit.bme.hu/ jakab/edu/litr/TimeSensNet/TSN-Time-Sensitive-Networking-White-Paper.pdf, 2016. [Online; Accessed 20.04.18].

[52] I. I. Consortium, "Time sensitive networking - flexible manufacturing." https://www.iiconsortium.org/pdf/TSN-brochure-2017-11-7.pdf, 2017. [Online; Accessed 25.04.18].

[53] I. I. Consortium, "About us." https://www.iiconsortium.org/about-us.htm, 2018. [Online; Accessed 20.04.18].

[54] C. systems, "Time-sensitive networking: A technical introduction." https://www.cisco.com/c/dam/en/us/solutions/collateral/industry-solutions/white-paper-c11-738950.pdf, 2017. [Online; Accessed 08.04.18].

Appendix A

Acronyms

6TiSCH	IPv6 over the TSCH mode of IEEE 802.15.4e
ACK	Acknowledgement
AP	Alternative Parent
APP	Application Layer
App-flow	Application-flow
ASN	Absolute Slot Number
BER	Bit Error Rate
C	Coulombs
CDT	Control-Data Traffic
CNC	Central Network Controller
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
CUC	Centralized User Configuration
CV	Coefficient of variation
DAG	Directed Acyclic Graph
DAO	Destination Advertisement Object
DetNet	Deterministic Network Working Group

DODAG	Destination Oriented Directed Acyclic Graph
DP	Default Parent
EB	Enhanced Beacon
FDMA	Frequency Division Multiple Access
FES	Future Event Set
HSR	High-Availability Seamless Redundancy
IE	Information Element
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IIoT	Industrial Internet of Things
IoT	Internet of Things
IP	Internet Protocol
IPv6	Internet Protocol version 6
IT	Informational Technology
LFC	Leapfrog Collaboration
MAC	Medium Access Control
MSF	Minimum Scheduling Function
OT	Operational Technology
PCE	Path Computation Element
PDR	Packet Delivery Ratio
PEF	Packet Elimination Function
PER	Packet Error Rate
POF	Packet Ordering Function
PRF	Packet Replication Function

PRP	Parallel Redundancy Protocol
QoS	Quality of Service
RF	Radio Frequency
RPE	Reverse Packet Elimination
RSSI	Received Signal Strength Indication
PTP	Precision Time Protocol
RX	Receiving
SINR	Signal-to-Interference-Plus-Noise Ratio
TDMA	Time Division Multiple Access
TSN	Time-Sensitive Networking
TSCH	Time-Slotted Channel Hopping
TX	Transmission
UUID	Universal Unique Identifier
WSN	Wireless Sensor Networks

Appendix B

Time-Sensitive Networking

IETF has in the draft "Deterministic Networking Use Cases" determined demands needed in an industrial IP network [31]. Moreover, DetNet defining layer 3 operations, and Time-Sensitive Networking (TSN) [25] is reshaping layer 2. Background for TSN working group is criteria's asked by the industry for machine-to-machine communication. Industry asks for a converged IP based network with deterministic behavior. Latency and jitter should be bounded. Availability should be high, presumably through redundancy. Typical numbers adopted state availability should be as high as 99.999%. Next, low message delivery time, around 100μ-50 ms. Packet loss is asked to be burst less and as low as 0.1-1%. Ntwork should be secure, e.g. prevent critical flows from being leaked between physically separated networks.

B.0.1 Current state

As of the current 2018, TSN is not a single standard document, but a collection of standards which is under development by IEEE 801.1 TSN Task Group [25] since 2012 [49]. TSN aims to provide deterministic capabilities to Ethernet. It is a layer 2 technology and not an IP standard. Forwarding decisiona are made on a TSN enabled device using the Ethernet header, not an IP address. All TSN devices are required to share a common knowledge of time. Prior to TSN, standard Ethernet did not have pure layer 2 deterministic abilities. Industries such as aerospace, automotive and manufacturing can benefit from TSN [50]. The industry has as a collective decided to work together on creating a standard, instead of working on separate proprietary solutions. The network was considered a noncompetitive zone, and vendors decided to compete on products and functionality. Vendors and developers has formed two different organizations: Avnu Alliance and the Industrial Internet Consortium to ensure the industry's interests in TSN development.

B.0.2 Avnu Alliance and Industrial Internet Consortium

These two organizations were established by the industry to ensure quality and multi-vendor support capabilities. Avnu Alliance was established to verify TSN and AVB products [51]. Manufacturers may use the Avnu Alliance logo on their products if they are certified for TSN or AVB interoperability. The Industrial Internet Consortium was established to ensure development of architectures that simplify multi-vendor systems and solutions [52]. Founding partners of Avnu Alliance and the Industrial Internet Consortium include Cisco, Intel and other large manufacturers [51, 53].

B.0.3 Time-Sensitive Networking standards

As mentioned earlier TSN is currently not a single standard but made up of a collection of standards. Table B.1 lists current TSN standards [52]. Other standards are under revision.

Standard	**Description**
IEEE 802.1 AS-REV	Timing and synchronization
IEEE 1588	Timing and synchronization
IEEE 802.1 Qbu	Frame Preemption
IEEE 802.3 br	Frame Preemption
IEEE 802.1 Qbv	Enhancements for scheduled traffic
IEEE 802.1 Qca	Path control and reservation
IEEE 802.1 Qcc	System configuration
IEEE 802.1 Qci	Per-stream filtering and policing
IEEE 802.1 CB	Seamless redundancy

Table B.1: List of TSN standards

B.0.4 TSN Components

This section describe the whitepaper Cisco has about TSN [54]. In addition there are different whitepapers but the Cisco whitepaper is presented. This is due to Cisco being a leading vendor in the networking industry.

Components Cisco describes five main components in their TSN description: TSN flow, end devices, bridges, Central Network Controller (CNC), and Centralized User Configuration (CUC). TSN flow is a term used to describe time-critical communication between end devices. Network devices uniquely identify each TSN flow. Network devices honor each flow strict time requirements. Next, end devices

are source and destination of TSN flows. These end devices run an application requiring deterministic communication. Source and destinations can also be referred to as talkers and listeners. Bridges can be referred to as Ethernet switches. But for TSN, these switches have special capabilities allowing them to transmit Ethernet frames on a TSN flow according to a schedule. TSN bridges receives Ethernet frames of a TSN flow according to a schedule. CNC acts as a proxy for the TSN enabled network. Specifically, it acts as a proxy for the bridges and their interconnections. CNC controls applications that require deterministic communication. TSN frames are transmitted on a schedule defined by CNC. In general, vendor of the device provides CNC application. In addition, CUC is provided by the vendor of the TSN. CUC represents the control applications and the end devices. CNC receives requests from CUC for deterministic TSN flows with specific requirements for given flows. Cisco uses Precision Time Protocol (PTP) to maintain common knowledge of time. Furthermore, PTP protocol versions utilized by TSN are IEEE 802.1AS and IEEE 802.1ASRev. Lastly, IEEE 802.1 introduce a new traffic class named Control-Data Traffic (CDT). This class is assigned highest priority to assure required guarantees.

Appendix C

6TiSCH Simulator Code

This Appendix contains all the python files that had to be adjusted for RPE. Only the methods, classes or lines that has been altered is presented. The original code can be located at: *https://bitbucket.org/6tisch/simulator/*

C.1 Path Computation Element

```
import MoteDefines as d
import random
import SimEngine
from random import randint
class PCE(object):

    TRACK_CELL_NUM = 1

    def __init__(self, mote):
        self.mote   = mote
        self.engine = SimEngine.SimEngine.SimEngine()
        self.log    = SimEngine.SimLog.SimLog().log
        self.engine.scheduleAtAsn(
            asn = 1,
            cb = self._build_track,
            uniqueTag = 'build tracks',
            intraSlotOrder = d.INTRASLOTORDER_STARTSLOT
        )
        #INDEX IS FROM 0-100 (101)
        self.slotA = 0
        self.slotB = 1 #ADJUST THE DAISY CHAIN
        self.slotRepA = 5
        self.slotRepB = 6 #DAISY CHAIN DELAY + 5

        '''self.slotA = 0
        self.slotB = 99 #ADJUST THE DAISY CHAIN
        self.slotRepA = 4
        self.slotRepB = 5 #DAISY CHAIN DELAY + 5'''

        '''self.slotA = 0
        self.slotB = 50 #ADJUST THE DAISY CHAIN
        self.slotRepA = 4
```

```
        self.slotRepB = 54 #DAISY CHAIN DELAY + 5'''

    def receive_request(self, packet):
        pass


    def _build_track(self):
        # build track from source mote to sink
        path_a = [7, 5, 3, 1 ,0]
        path_b = [7, 6, 4, 2, 0]
        reverse_path_a = [0, 2, 4, 6, 7]
        reverse_path_b = [0, 1, 3, 5, 7]
        self.computetrack_a(path_a)
        self.computetrack_reverse_a(reverse_path_a)
        self.computetrack_b(path_b)
        self.computetrack_reverse_b(reverse_path_b)




    def computetrack_a(self, path):
        for idx, moteid in enumerate(path):
            self.slotA += 1
            channel = randint(0, 15)
            if idx >= len(path) - 1:
             break
            sender = self.engine.motes[moteid]
            receiver = self.engine.motes[path[idx + 1]]

            sender.tsch.addCell(
                slotOffset       = self.slotA,
                channelOffset    = channel,
                neighbor         = receiver.get_mac_addr(),
                cellOptions      = [ d.CELLOPTION_TX ],
                trackID          = 0,
                slotframe_handle = 0,
            )
            receiver.tsch.addCell(
                slotOffset       = self.slotA,
                channelOffset    = channel,
                neighbor         = sender.get_mac_addr(),
                cellOptions      = [ d.CELLOPTION_RX ],
                trackID          = 0,
                slotframe_handle = 0
            )

    def computetrack_b(self, path):
        for idx, moteid in enumerate(path):
            if self.slotB >= 100:
                self.slotB = 0
            self.slotB += 1
            channel = randint(0, 15)

            if idx >= len(path) - 1:
             break
            sender = self.engine.motes[moteid]
            receiver = self.engine.motes[path[idx + 1]]

            sender.tsch.addCell(
                slotOffset       = self.slotB,
                channelOffset    = channel,
                neighbor         = receiver.get_mac_addr(),
```

```
            cellOptions      = [ d.CELLOPTION_TX ],
            trackID          = 0,
            slotframe_handle = 0,
        )

        receiver.tsch.addCell(
            slotOffset       = self.slotB,
            channelOffset    = channel,
            neighbor         = sender.get_mac_addr(),
            cellOptions      = [ d.CELLOPTION_RX ],
            trackID          = 0,
            slotframe_handle = 0
        )

def computetrack_reverse_a(self, path):
    for idx, moteid in enumerate(path):
        if self.slotRepA >= 100:
            self.slotRepA = 1
        self.slotRepA += 1
        channel = randint(0, 15)

        if idx >= len(path) - 1:
         break
        sender = self.engine.motes[moteid]
        receiver = self.engine.motes[path[idx + 1]]

        sender.tsch.addCell(
            slotOffset       = self.slotRepA,
            channelOffset    = channel,
            neighbor         = receiver.get_mac_addr(),
            cellOptions      = [ d.CELLOPTION_TX ],
            trackID          = 0,
            slotframe_handle = 0,
        )

        receiver.tsch.addCell(
            slotOffset       = self.slotRepA,
            channelOffset    = channel,
            neighbor         = sender.get_mac_addr(),
            cellOptions      = [ d.CELLOPTION_RX ],
            trackID          = 0,
            slotframe_handle = 0
        )
def computetrack_reverse_b(self, path):
    for idx, moteid in enumerate(path):
        if self.slotRepB >= 100:
            self.slotRepB = 0
        self.slotRepB += 1
        channel = randint(0, 15)

        if idx >= len(path) - 1:
         break
        sender = self.engine.motes[moteid]
        receiver = self.engine.motes[path[idx + 1]]

        sender.tsch.addCell(
            slotOffset       = self.slotRepB,
            channelOffset    = channel,
            neighbor         = receiver.get_mac_addr(),
            cellOptions      = [ d.CELLOPTION_TX ],
            trackID          = 0,
            slotframe_handle = 0,
```

```
        )

        receiver.tsch.addCell(
            slotOffset        = self.slotRepB,
            channelOffset     = channel,
            neighbor          = sender.get_mac_addr(),
            cellOptions       = [ d.CELLOPTION_RX ],
            trackID           = 0,
            slotframe_handle  = 0
        )
```

C.2 Applayer

```
class AppRoot(AppBase):
    """Handle application packets from motes
    """

    def __init__(self, mote):
        super(AppRoot, self).__init__(mote)
        self.pce = pce.PCE(self.mote)
    #=========================== public ==========================================


    def _send_replication_path_a(self, dstIp, packet_length, sequencenumber):

        # abort if I'm not ready to send DATA yet
        if self.mote.clear_to_send_EBs_DATA()==False:
            return

        # create
        packet = self._generate_packet(
            dstIp          = dstIp,
            packet_type    = d.PKT_TYPE_ACK_A,
            packet_length  = packet_length,
            sequencenumber = sequencenumber
        )
        #packet['app']['sequencenumber'] = sequencenumber

        # log
        self.log(
            SimEngine.SimLog.LOG_APP_TX,
            {
                '_mote_id':         self.mote.id,
                'packet':           packet,
            }
        )

        # send

        self.mote.sixlowpan.sendPacket(packet)

    def _send_replication_path_b(self, dstIp, packet_length, sequencenumber):

        # abort if I'm not ready to send DATA yet
        if self.mote.clear_to_send_EBs_DATA()==False:
            return

        # create
        packet = self._generate_packet(
            dstIp          = dstIp,
```

```
            packet_type     = d.PKT_TYPE_ACK_B,
            packet_length   = packet_length,
            sequencenumber = sequencenumber
        )

        # log
        self.log(
            SimEngine.SimLog.LOG_APP_TX,
            {
                '_mote_id':         self.mote.id,
                'packet':           packet,
            }
        )
        # send
        self.mote.sixlowpan.sendPacket(packet)

    def startSendingData(self):
        # nothing to schedule
        pass

    def recvPacket(self, packet):
        assert self.mote.dagRoot
        foundAck = False
        if packet['type'] == d.PKT_TYPE_PATH_A and packet['mac']['srcMac'] == self
↪ .engine.motes[1].get_mac_addr() or packet['type'] == d.PKT_TYPE_PATH_B and
↪ packet['mac']['srcMac'] == self.engine.motes[2].get_mac_addr():
            for Qpacket in self.mote.tsch.txQueue:
                if Qpacket['type'] == d.PKT_TYPE_ACK_A or Qpacket['type'] == d.
↪ PKT_TYPE_ACK_B:
                    if packet['app']['sequencenumber'] == Qpacket['app']['
↪ sequencenumber']:
                        self.mote.tsch.dequeue(Qpacket)
                        self.sinkdrops += 1
                        print 'Dropped at Sink:', Qpacket['type']
                        foundAck = True

        if not foundAck:
            if packet['type'] == d.PKT_TYPE_PATH_A and packet['mac']['srcMac'] ==
↪ self.engine.motes[1].get_mac_addr():

                self._send_rpe_b(packet['app']['sequencenumber'])
                print 'Sequencenumber A at Sink: ', packet['app']['sequencenumber
↪ ']


            elif packet['type'] == d.PKT_TYPE_PATH_B and packet['mac']['srcMac']
↪ == self.engine.motes[2].get_mac_addr():
                self._send_rpe_a(packet['app']['sequencenumber'])
                print 'Sequencenumber B at Sink: ', packet['app']['sequencenumber
↪ ']

            else:
                print 'This should not happen'


        # log and update mote stats

        self.log(
            SimEngine.SimLog.LOG_APP_RX,
            {
                '_mote_id': self.mote.id,
                'packet'  : packet
```

```
            }
        )

    #========================== private ==========================================


    def _send_rpe_a(self, sequencenumber, packet_length=None):

        if packet_length is None:
            packet_length = self.APP_PK_LENGTH

        self._send_replication_path_a(
            dstIp          = self.engine.motes[7].get_ipv6_global_addr(),
            sequencenumber = sequencenumber,
            packet_length  = 3

        )

        self.log(
            SimEngine.SimLog.LOG_REPL_TX,
            {
                '_mote_id': self.mote.id
            }
        )
    def _send_rpe_b(self, sequencenumber, packet_length=None):

        if packet_length is None:
            packet_length = self.APP_PK_LENGTH

        self._send_replication_path_b(
            dstIp          = self.engine.motes[7].get_ipv6_global_addr(),
            sequencenumber = sequencenumber,
            packet_length  = 3

        )

        self.log(
            SimEngine.SimLog.LOG_REPL_TX,
            {
                '_mote_id': self.mote.id
            }
        )


class AppPRE(AppBase):

    """Send a packet periodically

    The first timing to send a packet is randomly chosen between [next
    asn, (next asn + pkPeriod)].
    """

    def __init__(self, mote, **kwargs):
        super(AppPRE, self).__init__(mote)
        self.sending_first_packet = True

    #========================== public ==========================================

    def startSendingData(self):
        if self.sending_first_packet:
```

```
        self._schedule_transmission()

def recvPacket(self, packet):
    if self.mote.id == 7:
         if packet['type'] == d.PKT_TYPE_ACK_A:
            self.StopSendingA = False
            print 'A stopped'
         elif  packet['type'] == d.PKT_TYPE_ACK_B:
            self.StopSendingB = False
            print 'B stopped'
    pass


#========================== public ==========================================


def _schedule_transmission(self):

    assert self.settings.app_pkPeriod >= 0
    if self.settings.app_pkPeriod == 0:
        return

    if self.sending_first_packet:
        # compute initial time within the range of [next asn, next asn+
↪ pkPeriod]
        delay = self.settings.tsch_slotDuration + (self.settings.app_pkPeriod
↪ * random.random())
        self.sending_first_packet = False
    else:
        # compute random delay
        assert self.settings.app_pkPeriodVar < 1
        #delay = self.settings.app_pkPeriod * (1 + random.uniform(-self.
↪ settings.app_pkPeriodVar, self.settings.app_pkPeriodVar))
        delay = 10
    # schedule
    self.engine.scheduleAtAsn(
        asn             = self.engine.asn + 1,
        cb              = self._send_a_single_packet,
        uniqueTag       = (
            'AppPRE',
            'scheduled_by_{0}'.format(self.mote.id)
        ),
        intraSlotOrder  = d.INTRASLOTORDER_STARTSLOT,
    )

def _send_a_single_packet(self):
    waitSlotframes = 10
    slotframe = 101
    converged = 100000
    sendpackets = 4000
    if self.mote.rpl.dodagId == None:
        # it seems we left the dodag; stop the transmission
        self.sending_first_packet = True
        return
        #wait until 1000 slotframes have passed tp sync
    if  self.mote.id == 7 and self.engine.asn > converged:

        isNewSlotFrame = (self.engine.asn % (slotframe*waitSlotframes)) == 0
        if isNewSlotFrame:

            #Change parent at source
```

```
            if self.StopSendingB == True:
                self.engine.motes[7].rpl.of.set_preferred_parent(self.engine.
↪ motes[5].get_mac_addr())
                self._send_path_a(
                dstIp          = self.mote.rpl.dodagId,
                packet_length  = self.settings.app_pkLength
                )
                self.txpacket += 1


            if self.StopSendingA == True:
                self.engine.motes[7].rpl.of.set_preferred_parent(self.engine.
↪ motes[6].get_mac_addr())
                self._send_path_b(
                dstIp          = self.mote.rpl.dodagId,
                packet_length  = self.settings.app_pkLength
                )
                self.engine.motes[7].rpl.of.set_preferred_parent(self.engine.
↪ motes[5].get_mac_addr())
                self.txpacket += 1


     # schedule the next transmission
    if self.txpacket < sendpackets:
        self.StopSendingB = True
        self.StopSendingA = True
        self._schedule_transmission()
    else:
        pass

def _send_path_a(self, dstIp, packet_length):

    # abort if I'm not ready to send DATA yet
    if self.mote.clear_to_send_EBs_DATA()==False:
        return
    self.sequencenumber += 1
    self.appcounter += 1
    # create
    packet = self._generate_packet(
        dstIp          = dstIp,
        packet_type    = d.PKT_TYPE_PATH_A,
        packet_length  = packet_length,
        sequencenumber = self.sequencenumber
    )
    # log
    self.log(
        SimEngine.SimLog.LOG_APP_TX,
        {
            '_mote_id':         self.mote.id,
            'packet':           packet,
        }
    )
    packet['net']['downward'] = True
    # send
    print 'Sequence number A at initalization: ', packet['app']['
↪ sequencenumber']
    self.mote.sixlowpan.sendPacket(packet)



def _send_path_b(self, dstIp, packet_length):
```

```
        # abort if I'm not ready to send DATA yet
        if self.mote.clear_to_send_EBs_DATA()==False:
            return
        # create
        packet = self._generate_packet(
            dstIp          = dstIp,
            packet_type    = d.PKT_TYPE_PATH_B,
            packet_length  = packet_length,
            sequencenumber = self.sequencenumber
        )

        # log
        self.log(
            SimEngine.SimLog.LOG_APP_TX,
            {
                '_mote_id':        self.mote.id,
                'packet':          packet,
            }
        )

        # send
        print 'Sequence number B at initalization: ', packet['app']['
↪ sequencenumber']
        self.mote.sixlowpan.sendPacket(packet)


    def _send_replication_path_a(self, dstIp, packet_length, sequencenumber):

        # abort if I'm not ready to send DATA yet
        if self.mote.clear_to_send_EBs_DATA()==False:
            return

        # create
        packet = self._generate_packet(
            dstIp          = dstIp,
            packet_type    = d.PKT_TYPE_ACK_A,
            packet_length  = packet_length,
            sequencenumber = sequencenumber
        )
        packet['net']['downward'] = True
        # log
        self.log(
            SimEngine.SimLog.LOG_APP_TX,
            {
                '_mote_id':        self.mote.id,
                'packet':          packet,
            }
        )

        # send

        self.mote.sixlowpan.sendPacket(packet)

    def _send_replication_path_b(self, dstIp, packet_length, sequencenumber):

        # abort if I'm not ready to send DATA yet
        if self.mote.clear_to_send_EBs_DATA()==False:
            return

        # create
        packet = self._generate_packet(
```

```
            dstIp           = dstIp,
            packet_type     = d.PKT_TYPE_ACK_B,
            packet_length   = packet_length,
            sequencenumber = sequencenumber
        )
        packet['net']['downward'] = True
        # log
        self.log(
            SimEngine.SimLog.LOG_APP_TX,
            {
                '_mote_id':         self.mote.id,
                'packet':           packet,
            }
        )

        # send

        self.mote.sixlowpan.sendPacket(packet)
```

C.3 AppLayer - Long delay

```
class AppRoot(AppBase):
    """Handle application packets from motes
    """

    def __init__(self, mote):
        super(AppRoot, self).__init__(mote)
        self.pce = pce.PCE(self.mote)
    #========================= public ==========================================


    def _send_replication_path_a(self, dstIp, packet_length, sequencenumber):

        # abort if I'm not ready to send DATA yet
        if self.mote.clear_to_send_EBs_DATA()==False:
            return

        # create
        packet = self._generate_packet(
            dstIp           = dstIp,
            packet_type     = d.PKT_TYPE_ACK_A,
            packet_length   = packet_length,
            sequencenumber = sequencenumber
        )
        #packet['app']['sequencenumber'] = sequencenumber

        # log
        self.log(
            SimEngine.SimLog.LOG_APP_TX,
            {
                '_mote_id':         self.mote.id,
                'packet':           packet,
            }
        )

        # send

        self.mote.sixlowpan.sendPacket(packet)

    def _send_replication_path_b(self, dstIp, packet_length, sequencenumber):
```

```
        # abort if I'm not ready to send DATA yet
        if self.mote.clear_to_send_EBs_DATA()==False:
            return

        # create
        packet = self._generate_packet(
            dstIp          = dstIp,
            packet_type    = d.PKT_TYPE_ACK_B,
            packet_length  = packet_length,
            sequencenumber = sequencenumber
        )

        # log
        self.log(
            SimEngine.SimLog.LOG_APP_TX,
            {
                '_mote_id':         self.mote.id,
                'packet':           packet,
            }
        )
        # send
        self.mote.sixlowpan.sendPacket(packet)

    def startSendingData(self):
        # nothing to schedule
        pass

    def recvPacket(self, packet):
        assert self.mote.dagRoot
        foundAck = False
        if packet['type'] == d.PKT_TYPE_PATH_A and packet['mac']['srcMac'] == self
    ↪ .engine.motes[1].get_mac_addr() or packet['type'] == d.PKT_TYPE_PATH_B and
    ↪ packet['mac']['srcMac'] == self.engine.motes[2].get_mac_addr():
            for Qpacket in self.mote.tsch.txQueue:
                if Qpacket['type'] == d.PKT_TYPE_ACK_A or Qpacket['type'] == d.
    ↪ PKT_TYPE_ACK_B:
                    if packet['app']['sequencenumber'] == Qpacket['app']['
    ↪ sequencenumber']:
                        self.mote.tsch.dequeue(Qpacket)
                        self.sinkdrops += 1
                        print 'Dropped at Sink:' , Qpacket['type']
                        foundAck = True

        if not foundAck:
            if packet['type'] == d.PKT_TYPE_PATH_A and packet['mac']['srcMac'] ==
    ↪ self.engine.motes[1].get_mac_addr():
                self._send_rpe_b(packet['app']['sequencenumber'])
                print 'Sequencenumber A at Sink: ', packet['app']['sequencenumber
    ↪ ']


            elif packet['type'] == d.PKT_TYPE_PATH_B and packet['mac']['srcMac']
    ↪ == self.engine.motes[2].get_mac_addr():
                print 'Sequencenumber B at Sink: ', packet['app']['sequencenumber
    ↪ ']

            else:
                print 'This should not happen'


        # log and update mote stats
```

```
        self.log(
            SimEngine.SimLog.LOG_APP_RX,
            {
                '_mote_id': self.mote.id,
                'packet'  : packet
            }
        )

    #========================== private =========================================


    def _send_rpe_a(self, sequencenumber, packet_length=None):

        if packet_length is None:
            packet_length = self.APP_PK_LENGTH

        self._send_replication_path_a(
            dstIp          = self.engine.motes[7].get_ipv6_global_addr(),
            sequencenumber = sequencenumber,
            packet_length  = 23

        )

        self.log(
            SimEngine.SimLog.LOG_REPL_TX,
            {
                '_mote_id': self.mote.id
            }
        )
    def _send_rpe_b(self, sequencenumber, packet_length=None):

        if packet_length is None:
            packet_length = self.APP_PK_LENGTH

        self._send_replication_path_b(
            dstIp          = self.engine.motes[7].get_ipv6_global_addr(),
            sequencenumber = sequencenumber,
            packet_length  = 23

        )

        self.log(
            SimEngine.SimLog.LOG_REPL_TX,
            {
                '_mote_id': self.mote.id
            }
        )


class AppPRE(AppBase):

    """Send a packet periodically

    The first timing to send a packet is randomly chosen between [next
    asn, (next asn + pkPeriod)].
    """

    def __init__(self, mote, **kwargs):
        super(AppPRE, self).__init__(mote)
```

```
        self.sending_first_packet = True
        self.StopSendingA = False

#======================== public ==========================================

def startSendingData(self):
    if self.sending_first_packet:
        self._schedule_transmission()

def recvPacket(self, packet):
    if self.mote.id == 7 and packet['type'] == d.PKT_TYPE_ACK_B:
        print 'PATH A Returned'
        sequencenumber = packet['app']['sequencenumber']
        self.engine.removeFutureEvent("criticalpacket_b" + str(sequencenumber)
↪ )
    else:
        pass



#======================== public ==========================================


def _schedule_transmission(self):

    assert self.settings.app_pkPeriod >= 0
    if self.settings.app_pkPeriod == 0:
        return

    if self.sending_first_packet:
        # compute initial time within the range of [next asn, next asn+
↪ pkPeriod]
        delay = self.settings.tsch_slotDuration + (self.settings.app_pkPeriod
↪ * random.random())
        self.sending_first_packet = False
    else:
        # compute random delay
        assert self.settings.app_pkPeriodVar < 1
        #delay = self.settings.app_pkPeriod * (1 + random.uniform(-self.
↪ settings.app_pkPeriodVar, self.settings.app_pkPeriodVar))
        delay = 10
    # schedule
    self.engine.scheduleAtAsn(
        asn             = self.engine.asn + 1,
        cb              = self._send_a_single_packet,
        uniqueTag       = (
            'AppPRE',
            'scheduled_by_{0}'.format(self.mote.id)
        ),
        intraSlotOrder  = d.INTRASLOTORDER_STARTSLOT,
    )

def _send_a_single_packet(self):
        converged = 100000
        sendpackets = 4000
        if self.mote.rpl.dodagId == None:
            # it seems we left the dodag; stop the transmission
            self.sending_first_packet = True
            return
            #wait until 1000 slotframes have passed tp sync
        if  self.mote.id == 7 and self.engine.asn > converged:
```

```
            if  self.engine.asn % 1010 == 0:
                self.engine.motes[7].rpl.of.set_preferred_parent(self.engine.
↪ motes[5].get_mac_addr())
                self._send_path_a(
                    dstIp          = self.mote.rpl.dodagId,
                    packet_length  = self.settings.app_pkLength
                )
                self.txpacket += 1
                self.engine.scheduleAtAsn(
                    asn            = self.engine.getAsn() + (101*16), # sett
↪ antal slotframes  vente her
                    cb             = self._sendB,
                    uniqueTag      = "criticalpacket_b" + str(self.
↪ sequencenumber),
                    intraSlotOrder = d.INTRASLOTORDER_ADMINTASKS
                )



        # schedule the next transmission
        if self.txpacket < sendpackets:
            self._schedule_transmission()

        else:
            pass


def _sendB(self):
    self.engine.motes[7].rpl.of.set_preferred_parent(self.engine.motes[6].
↪ get_mac_addr())
    self._send_path_b(
        dstIp          = self.mote.rpl.dodagId,
        packet_length  = self.settings.app_pkLength
    )
    self.engine.motes[7].rpl.of.set_preferred_parent(self.engine.motes[5].
↪ get_mac_addr())
    self.txpacket += 1




def _send_path_a(self, dstIp, packet_length):

    # abort if I'm not ready to send DATA yet
    if self.mote.clear_to_send_EBs_DATA()==False:
        return
    self.sequencenumber += 1
    self.appcounter += 1
    # create
    packet = self._generate_packet(
        dstIp          = dstIp,
        packet_type    = d.PKT_TYPE_PATH_A,
        packet_length  = packet_length,
        sequencenumber = self.sequencenumber
    )
    # log
    self.log(
        SimEngine.SimLog.LOG_APP_TX,
```

```
            {
                '_mote_id':         self.mote.id,
                'packet':           packet,
            }
        )
        packet['net']['downward'] = True
        # send
        print 'Sequence number A at initalization: ', packet['app']['
↪ sequencenumber']
        self.mote.sixlowpan.sendPacket(packet)


    def _send_path_b(self, dstIp, packet_length):

        # abort if I'm not ready to send DATA yet
        if self.mote.clear_to_send_EBs_DATA()==False:
            return
        # create
        packet = self._generate_packet(
            dstIp          = dstIp,
            packet_type    = d.PKT_TYPE_PATH_B,
            packet_length  = packet_length,
            sequencenumber = self.sequencenumber - 1
        )

        # log
        self.log(
            SimEngine.SimLog.LOG_APP_TX,
            {
                '_mote_id':         self.mote.id,
                'packet':           packet,
            }
        )

        # send
        print 'Sequence number B at initalization: ', packet['app']['
↪ sequencenumber']
        self.mote.sixlowpan.sendPacket(packet)


    def _send_replication_path_a(self, dstIp, packet_length, sequencenumber):

        # abort if I'm not ready to send DATA yet
        if self.mote.clear_to_send_EBs_DATA()==False:
            return

        # create
        packet = self._generate_packet(
            dstIp          = dstIp,
            packet_type    = d.PKT_TYPE_ACK_A,
            packet_length  = packet_length,
            sequencenumber = sequencenumber
        )
        packet['net']['downward'] = True
        # log
        self.log(
            SimEngine.SimLog.LOG_APP_TX,
            {
                '_mote_id':         self.mote.id,
                'packet':           packet,
            }
```

```
        )

        # send

        self.mote.sixlowpan.sendPacket(packet)

    def _send_replication_path_b(self, dstIp, packet_length, sequencenumber):

        # abort if I'm not ready to send DATA yet
        if self.mote.clear_to_send_EBs_DATA()==False:
            return

        # create
        packet = self._generate_packet(
            dstIp          = dstIp,
            packet_type    = d.PKT_TYPE_ACK_B,
            packet_length  = packet_length,
            sequencenumber = sequencenumber
        )
        packet['net']['downward'] = True
        # log
        self.log(
            SimEngine.SimLog.LOG_APP_TX,
            {
                '_mote_id':        self.mote.id,
                'packet':          packet,
            }
        )

        # send

        self.mote.sixlowpan.sendPacket(packet)
```

C.4 RPL

```
class RplOFStatic(object):

    def __init__(self, rpl):
        self.rpl = rpl
        self.rank = None
        self.preferred_parent = None


    def update(self, dio):
        if self.preferred_parent is None and self.rpl.mote.id != 0:
            parent_mote = None
            if self.rpl.mote.id == 1:
                parent_mote = self.rpl.engine.motes[0]
                self.rank = 512
            elif self.rpl.mote.id == 2:
                parent_mote = self.rpl.engine.motes[0]
                self.rank = 512
            elif self.rpl.mote.id == 3:
                parent_mote = self.rpl.engine.motes[1]
                self.rank = 1024
            elif self.rpl.mote.id == 4:
                parent_mote = self.rpl.engine.motes[2]
                self.rank = 1024
            elif self.rpl.mote.id == 5:
                parent_mote = self.rpl.engine.motes[3]
```

```
            self.rank = 1536
        elif self.rpl.mote.id == 6:
            parent_mote = self.rpl.engine.motes[4]
            self.rank = 1536
        elif self.rpl.mote.id == 7:
            parent_mote = self.rpl.engine.motes[6]
            self.rank = 2048
        #Check if TSCH is synced
        if parent_mote is not None and parent_mote.tsch.getIsSync() :
            self.preferred_parent = parent_mote.get_mac_addr()
            self.rpl.indicate_preferred_parent_change(
                old_preferred = None,
                new_preferred = self.preferred_parent
            )

    def set_rank(self, new_rank):
        self.rank = new_rank

    def set_preferred_parent(self, new_preferred_parent):
        self.preferred_parent = new_preferred_parent

    def get_preferred_parent(self):
        return self.preferred_parent


    def update_etx(self, cell, mac_addr, isACKed):
        # do nothing
        pass
```

C.5 Scheduling function

```
    def _get_available_slots_global(self):
        busyslots = []
        for mote in self.engine.motes:
            for key in mote.tsch.slotframes:
                slots = mote.tsch.slotframes[key].get_busy_slots()
                for slot in slots:
                    busyslots.append(slot)
        return list((set(range(self.engine.settings.tsch_slotframeLength))) - set(
    ↪ busyslots))

    def _get_autonomous_cell(self, mac_addr):
        return self.engine.get_mote_by_mac_addr(mac_addr).sf.autonomous_cell

    def _allocate_autonomous_rx_cell(self):
        all_slots = self._get_available_slots_global()
        selected_slot = random.choice(all_slots)
        channel_offset = random.randint(0, 15)
        self.mote.tsch.addCell(
            slotOffset       = selected_slot,
            channelOffset    = channel_offset,
            neighbor         = None,
            cellOptions      = [d.CELLOPTION_RX],
            slotframe_handle = self.SLOTFRAME_HANDLE
        )
        self.autonomous_cell = (selected_slot, channel_offset)

    def _allocate_autonomous_tx_cell(self, mac_addr):
        slot_offset, channel_offset = self._get_autonomous_cell(mac_addr)
        self.mote.tsch.addCell(
```

```
            slotOffset       = slot_offset,
            channelOffset    = channel_offset,
            neighbor         = mac_addr,
            cellOptions      = [d.CELLOPTION_TX, d.CELLOPTION_SHARED],
            slotframe_handle = self.SLOTFRAME_HANDLE
        )
```

C.6 TSCH

```
    def get_first_packet_to_send(self, dst_mac_addr=None, trackID=None):
        packet_to_send = None
        if dst_mac_addr is None:
            if len(self.txQueue) == 0:
                # txQueue is empty; we may return an EB
                if (
                        self.mote.clear_to_send_EBs_DATA()
                        and
                        self._decided_to_send_eb()
                    ):
                    packet_to_send = self._create_EB()
                else:
                    packet_to_send = None
            else:
                # return the first one in the TX queue, whose destination MAC
                # is not associated with any of allocated (dedicated) TX cells
                for packet in self.txQueue:
                    packet_to_send = packet # tentatively
                    for _, slotframe in self.slotframes.items():
                        dedicated_tx_cells = filter(
                            lambda cell: d.CELLOPTION_TX in cell.options,
                            slotframe.get_cells_by_mac_addr(packet['mac']['dstMac
↪ ']))
                        )
                        if len(dedicated_tx_cells) > 0:
                            packet_to_send = None
                            break # try the next packet in TX queue

                    if packet_to_send is not None:
                        # found a good packet to send
                        break

                # if no suitable packet is found, packet_to_send remains None
        else:
            types = [d.PKT_TYPE_PATH_A, d.PKT_TYPE_PATH_B, d.PKT_TYPE_ACK_A, d.
↪ PKT_TYPE_ACK_B]
            for packet in self.txQueue:
                if packet['mac']['dstMac'] == dst_mac_addr:
                    if trackID is not None:
                        if packet['type'] in types:
                            packet_to_send = packet
                            break
                    else:
                        if packet['type'] not in types:
                            packet_to_send = packet
                            break


        return packet_to_send
```

C.7 Connectivity matrix

```
class ConnectivityMaster(ConnectivityBase):

    def _init_connectivity_matrix(self):
        for source in self.engine.motes:
            for destination in self.engine.motes:
                for channel in range(self.settings.phy_numChans):

                    #connectivity = self.CONNECTIVITY_MATRIX_NO_LINK
                    connectivity = self.CONNECTIVITY_MATRIX_NO_LINK

                    #DOWNSTREAM LINKS
                    if source.id == 0 and destination.id == 1:
                        connectivity = copy.copy(self.
↪ CONNECTIVITY_MATRIX_90_DOWNSTREAM_LINK)
                    elif source.id == 0 and destination.id == 2:
                        connectivity = copy.copy(self.
↪ CONNECTIVITY_MATRIX_90_DOWNSTREAM_LINK)
                    elif source.id == 1 and destination.id == 3:
                        connectivity = copy.copy(self.
↪ CONNECTIVITY_MATRIX_90_DOWNSTREAM_LINK)
                    elif source.id == 3 and destination.id == 5:
                        connectivity = copy.copy(self.
↪ CONNECTIVITY_MATRIX_90_DOWNSTREAM_LINK)
                    elif source.id == 5 and destination.id == 7:
                        connectivity = copy.copy(self.
↪ CONNECTIVITY_MATRIX_90_DOWNSTREAM_LINK)
                    elif source.id == 2 and destination.id == 4:
                        connectivity = copy.copy(self.
↪ CONNECTIVITY_MATRIX_90_DOWNSTREAM_LINK)
                    elif source.id == 4 and destination.id == 6:
                        connectivity = copy.copy(self.
↪ CONNECTIVITY_MATRIX_90_DOWNSTREAM_LINK)
                    elif source.id == 6 and destination.id == 7:
                        connectivity = copy.copy(self.
↪ CONNECTIVITY_MATRIX_90_DOWNSTREAM_LINK)
                    #UPSTREAM LINKS
                    elif source.id == 7 and destination.id == 5:
                        connectivity = copy.copy(self.CONNECTIVITY_MATRIX_90_LINK)
                    elif source.id == 7 and destination.id == 6:
                        connectivity = copy.copy(self.CONNECTIVITY_MATRIX_90_LINK)
                    elif source.id == 6 and destination.id == 4:
                        connectivity = copy.copy(self.CONNECTIVITY_MATRIX_90_LINK)
                    elif source.id == 4 and destination.id == 2:
                        connectivity = copy.copy(self.CONNECTIVITY_MATRIX_90_LINK)
                    elif source.id == 2 and destination.id == 0:
                        connectivity = copy.copy(self.CONNECTIVITY_MATRIX_90_LINK)
                    elif source.id == 5 and destination.id == 3:
                        connectivity = copy.copy(self.CONNECTIVITY_MATRIX_90_LINK)
                    elif source.id == 3 and destination.id == 1:
                        connectivity = copy.copy(self.CONNECTIVITY_MATRIX_90_LINK)
                    elif source.id == 1 and destination.id == 0:
                        connectivity = copy.copy(self.CONNECTIVITY_MATRIX_90_LINK)

                    #Leapfrog links
                    #Across
                    elif source.id == 1 and destination.id == 2 or source.id == 2
↪ and destination.id == 1:
                        connectivity = copy.copy(self.CONNECTIVITY_MATRIX_50_LINK)
                    elif source.id == 3 and destination.id == 4 or source.id == 4
```

```
↪ and destination.id == 3:
                    connectivity = copy.copy(self.CONNECTIVITY_MATRIX_50_LINK)
                elif source.id == 5 and destination.id == 6 or source.id == 6
↪ and destination.id == 5:
                    connectivity = copy.copy(self.CONNECTIVITY_MATRIX_50_LINK)
                #Diagonal
                elif source.id == 1 and destination.id == 4 or source.id == 4
↪ and destination.id == 1:
                    connectivity = copy.copy(self.CONNECTIVITY_MATRIX_70_LINK)
                elif source.id == 2 and destination.id == 3 or source.id == 3
↪ and destination.id == 2:
                    connectivity = copy.copy(self.CONNECTIVITY_MATRIX_70_LINK)
                elif source.id == 3 and destination.id == 6 or source.id == 6
↪ and destination.id == 3:
                    connectivity = copy.copy(self.CONNECTIVITY_MATRIX_70_LINK)
                elif source.id == 4 and destination.id == 5 or source.id == 5
↪ and destination.id == 4:
                    connectivity = copy.copy(self.CONNECTIVITY_MATRIX_70_LINK)


                self.connectivity_matrix[source.id][destination.id][channel] =
↪  copy.copy(
                    connectivity
                )


    assert self.connectivity_matrix[0][1][0]['rssi'] != -1000
```

C.8 6LoWPAN

```
def _find_nexthop_mac_addr(self, packet):
    mac_addr = None
    src_ip_addr = netaddr.IPAddress(packet['net']['srcIp'])
    dst_ip_addr = netaddr.IPAddress(packet['net']['dstIp'])
    # use lower 64 bits and invert U/L bit
    derived_dst_mac = str(
        netaddr.EUI(
            (int(dst_ip_addr) & 0xFFFFFFFFFFFFFFFF) ^ 0x0200000000000000
        )
    )

    #SET PATH
    if packet['type'] == 'DATA':
        mac_addr = self.mote.rpl.getPreferredParent()
        return mac_addr

    if packet['type'] == 'ACK_A' and self.mote.id == 0:
        return self.engine.motes[1].get_mac_addr()

    if packet['type'] == 'ACK_B' and self.mote.id == 0:
        return self.engine.motes[2].get_mac_addr()

    if packet['type'] == 'ACK_A' and self.mote.id == 1:
        return self.engine.motes[3].get_mac_addr()

    if packet['type'] == 'ACK_B' and self.mote.id == 2:
        return self.engine.motes[4].get_mac_addr()

    if packet['type'] == 'ACK_A' and self.mote.id == 3:
```

```
            return self.engine.motes[5].get_mac_addr()

        if packet['type'] == 'ACK_B' and self.mote.id == 4:
            return self.engine.motes[6].get_mac_addr()

        if packet['type'] == 'ACK_A' and self.mote.id == 5:
            return self.engine.motes[7].get_mac_addr()

        if packet['type'] == 'ACK_B' and self.mote.id == 6:
            return self.engine.motes[7].get_mac_addr()


        if (dst_ip_addr.words[0] & 0xFF00) == 0xFF00:
            # this is an IPv6 multicast address
            mac_addr = d.BROADCAST_ADDRESS

        elif self.mote.dagRoot:
            if derived_dst_mac in self.on_link_neighbor_list:
                # on-link
                mac_addr = derived_dst_mac
            else:
                # off-link
                mac_addr = None
        else:
            if self.mote.rpl.dodagId is None:
                print 'Not joined yet', self.mote.id
                # upward during secure join process
                mac_addr = str(self.mote.tsch.join_proxy)
            elif (
                    (
                        ((src_ip_addr.words[0] & 0xFE80) == 0xFE80)
                    )
                    or
                    (
                        ('downward' in packet['net'])
                        and
                        (packet['net']['downward'] is True)
                    )
                ):
                if derived_dst_mac in self.on_link_neighbor_list:
                    # on-link
                    mac_addr = derived_dst_mac
                else:
                    mac_addr = None
            else:
                # use the default router (preferred parent)
                mac_addr = self.mote.rpl.getPreferredParent()

        return mac_addr
```

These are snippets from each of the 6lowpan operations forward, recvPacket and sendPacket.

```
    def forward
            if rxPacket['type'] == d.PKT_TYPE_ACK_A or rxPacket['type'] == d.
    ↪ PKT_TYPE_ACK_B:
            for Qpacket in self.mote.tsch.txQueue:
                if Qpacket['type'] == d.PKT_TYPE_PATH_A or Qpacket['type'] == d.
    ↪ PKT_TYPE_PATH_B:
                    if rxPacket['app']['sequencenumber'] == Qpacket['app']['
    ↪ sequencenumber']:
                        self.mote.tsch.dequeue(Qpacket)
```

```
                    print 'dropped at' , self.mote.id , ' Packet' , Qpacket['
↪ type']
                    self.mote.drop_packet(
                            packet  = Qpacket,
                            reason  = SimEngine.SimLog.REVERSE_DROP,
                    )

                    goOn = False
    if rxPacket['type'] == d.PKT_TYPE_PATH_A or rxPacket['type'] == d.
↪ PKT_TYPE_PATH_B:
        for Qpacket in self.mote.tsch.txQueue:
            if Qpacket['type'] == d.PKT_TYPE_ACK_A or Qpacket['type'] == d.
↪ PKT_TYPE_ACK_B:
                if rxPacket['app']['sequencenumber'] == Qpacket['app']['
↪ sequencenumber']:
                    self.mote.tsch.dequeue(Qpacket)
                    print 'dropped at' , self.mote.id , ' Packet' , rxPacket['
↪ type']
                    self.mote.drop_packet(
                            packet  = rxPacket,
                            reason  = SimEngine.SimLog.REVERSE_DROP,
                    )
                    goOn = False
def recvPacket
        if packet['type'] == d.PKT_TYPE_ACK_A or packet['type'] == d.
↪ PKT_TYPE_ACK_B:
        for Qpacket in self.mote.tsch.txQueue:
            if Qpacket['type'] == d.PKT_TYPE_PATH_A or Qpacket['type'] == d.
↪ PKT_TYPE_PATH_B:
                if packet['app']['sequencenumber'] == Qpacket['app']['
↪ sequencenumber']:
                    self.mote.tsch.dequeue(Qpacket)
                    print 'dropped at' , self.mote.id , ' Packet' , Qpacket['
↪ type']
                    self.mote.drop_packet(
                            packet  = Qpacket,
                            reason  = SimEngine.SimLog.REVERSE_DROP,
                    )

                    goOn = False
    if self.mote.id == 0:
        if packet['type'] == d.PKT_TYPE_PATH_A or packet['type'] == d.
↪ PKT_TYPE_PATH_B:
            for Qpacket in self.mote.tsch.txQueue:
                if Qpacket['type'] == d.PKT_TYPE_ACK_A or Qpacket['type'] == d
↪ .PKT_TYPE_ACK_B:
                    if packet['app']['sequencenumber'] == Qpacket['app']['
↪ sequencenumber']:
                        self.mote.tsch.dequeue(Qpacket)
                        self.mote.tsch.dequeue(packet)
                        print 'Dropped at 6lowpan at Sink:' ,packet['type']
                        self.mote.drop_packet(
                            packet  = Qpacket,
                            reason  = SimEngine.SimLog.REVERSE_DROP,
                        )

                        goOn = False
def sendPacket
        if packet['type'] == d.PKT_TYPE_ACK_A or packet['type'] == d.
↪ PKT_TYPE_ACK_B:
        for Qpacket in self.mote.tsch.txQueue:
            if Qpacket['type'] == d.PKT_TYPE_PATH_A or Qpacket['type'] == d.
```

```
↪ PKT_TYPE_PATH_B:
                if packet['app']['sequencenumber'] == Qpacket['app']['
↪ sequencenumber']:
                    self.mote.tsch.dequeue(Qpacket)
                    print 'dropped at', self.mote.id, 'Packet', Qpacket['
↪ type']
                    self.mote.drop_packet(
                            packet  = Qpacket,
                            reason  = SimEngine.SimLog.REVERSE_DROP,
                    )

                    goOn = False
```

C.9 Mote

The adjustments to Mote.py to allow for starting with a converged network

```
def boot(self):
    self.tsch.join_proxy = netaddr.EUI(self.engine.motes[0].get_mac_addr())
    if self.dagRoot:
        # I'm the DAG root

        # app
        self.app.startSendingData()     # dagRoot
        # secjoin
        self.secjoin.setIsJoined(True)  # dagRoot
        self.tsch.clock.sync()
        self.tsch.setIsSync(True)       # dagRoot
        self.tsch.add_minimal_cell()    # dagRpot
        self.tsch.startSendingEBs()     # dagRoot
        # rpl
        self.rpl.start()
        # tsch

    else:
        # I'm NOT the DAG root

        # schedule the first listeningForE cell
        #self.tsch.schedule_next_listeningForEB_cell()
          # app
        # secjoin
        self.secjoin.setIsJoined(True)  # dagRoot
        # tsch
        if self.id == 1:
            self.tsch.join_proxy =  netaddr.EUI(self.engine.motes[0].
↪ get_mac_addr())
            self.tsch.clock.sync(self.engine.motes[0].get_mac_addr())
        elif self.id == 2:
            self.tsch.join_proxy =  netaddr.EUI(self.engine.motes[0].
↪ get_mac_addr())
            self.tsch.clock.sync(self.engine.motes[0].get_mac_addr())
        elif self.id == 3:
            self.tsch.join_proxy =  netaddr.EUI(self.engine.motes[1].
↪ get_mac_addr())
            self.tsch.clock.sync(self.engine.motes[1].get_mac_addr())
        elif self.id == 4:
            self.tsch.join_proxy =  netaddr.EUI(self.engine.motes[2].
↪ get_mac_addr())
            self.tsch.clock.sync(self.engine.motes[2].get_mac_addr())
        elif self.id == 5:
```

```
            self.tsch.join_proxy =  netaddr.EUI(self.engine.motes[3].
↪ get_mac_addr())
            self.tsch.clock.sync(self.engine.motes[3].get_mac_addr())
        elif self.id == 6:
            self.tsch.join_proxy =  netaddr.EUI(self.engine.motes[4].
↪ get_mac_addr())
            self.tsch.clock.sync(self.engine.motes[4].get_mac_addr())
        elif self.id == 7:
            self.tsch.join_proxy =  netaddr.EUI(self.engine.motes[5].
↪ get_mac_addr())
            self.tsch.clock.sync(self.engine.motes[5].get_mac_addr())
        self.tsch.setIsSync(True)
        self.tsch.add_minimal_cell()
        self.tsch.startSendingEBs()
          # rpl
        self.rpl.start()
```

Appendix D

Data retrival to CSV

This appendix shows how the data was retrieved from a json key performance indicators file and converted to a CSV file.

```
import json
import csv
import os
#os.chdir(r'C:/)
path = "C:/Users/Fredrik/Dropbox/sims/90 link late frame/exec_numMotes_8.dat.kpi"

dropsAtSink = []
dropsAtSink.append('Drops Mote 0')

dropsAtOne = []
dropsAtOne.append('Drops Mote 1')

dropsAtTwo = []
dropsAtTwo.append('Drops Mote 2')

dropsAtThree = []
dropsAtThree.append('Drops Mote 3')

dropsAtFour = []
dropsAtFour.append('Drops Mote 4')

dropsAtFive = []
dropsAtFive.append('Drops Mote 5')

dropsAtSix = []
dropsAtSix.append('Drops Mote 6')

dropsAtSeven = []
dropsAtSeven.append('Drops Mote 7')

battery_1 = []
battery_1.append('Lifetime Years Mote 1')

battery_2 = []
battery_2.append('Lifetime Years Mote 2')

battery_3 = []
battery_3.append('Lifetime Years Mote 3')
```

```
battery_4 = []
battery_4.append('Lifetime Years Mote 4')

battery_5 = []
battery_5.append('Lifetime Years Mote 5')

battery_6 = []
battery_6.append('Lifetime Years Mote 6')

battery_7 = []
battery_7.append('Lifetime Years Mote 7')

lowestlife = []
lowestlife.append('Lowest Battery Time')

appsent = []
appsent.append('Packets sent')

applost = []
applost.append('Packets lost')

appreceived = []
appreceived.append('Packets received')

max_latency = []
max_latency.append('Max Latency')

min_latency = []
min_latency.append('Min Latency')

mean_latencies = []
mean_latencies.append('Average Latency')

ratio = []
ratio.append('Packet delivery ratio')

ninty_nine = []
ninty_nine.append('99%')

with open(path, 'r') as json_file:
    data = json.load(json_file)
    for run in data:
        drop0 = data[run]['0']['packet_drops']['Elimination drop']
        dropsAtSink.append(drop0)

        drop1 = data[run]['1']['packet_drops']['Elimination drop']
        dropsAtOne.append(drop1)

        drop2 = data[run]['2']['packet_drops']['Elimination drop']
        dropsAtTwo.append(drop2)

        drop3 = data[run]['3']['packet_drops']['Elimination drop']
        dropsAtThree.append(drop3)

        drop4 = data[run]['4']['packet_drops']['Elimination drop']
        dropsAtFour.append(drop4)

        drop5 = data[run]['5']['packet_drops']['Elimination drop']
        dropsAtFive.append(drop5)

        drop6 = data[run]['6']['packet_drops']['Elimination drop']
        dropsAtSix.append(drop6)
```

```
        drop7 = data[run]['7']['packet_drops']['Elimination drop']
        dropsAtSeven.append(drop7)

        life1 = data[run]['1']['lifetime_AA_years']
        battery_1.append(life1)

        life2 = data[run]['2']['lifetime_AA_years']
        battery_2.append(life2)

        life3 = data[run]['3']['lifetime_AA_years']
        battery_3.append(life3)

        life4 = data[run]['4']['lifetime_AA_years']
        battery_4.append(life4)

        life5 = data[run]['5']['lifetime_AA_years']
        battery_5.append(life5)

        life6 = data[run]['6']['lifetime_AA_years']
        battery_6.append(life6)

        life7 = data[run]['7']['lifetime_AA_years']
        battery_7.append(life7)

        networkdies = data[run]['global-stats']['network_lifetime'][0]['min']
        lowestlife.append(networkdies)

        sent = data[run]['global-stats']['app-packets-sent'][0]['total']
        appsent.append(sent)

        lost = data[run]['global-stats']['app_packets_lost'][0]['total']
        applost.append(lost)

        received = data[run]['global-stats']['app_packets_received'][0]['total']
        appreceived.append(received)

        deliveryratio = data[run]['global-stats']['e2e-upstream-delivery'][0]['
    ↪ value']
        ratio.append(deliveryratio)

        minlatency = data[run]['global-stats']['e2e-upstream-latency'][0]['min']
        min_latency.append(minlatency)

        maxlatency = data[run]['global-stats']['e2e-upstream-latency'][0]['max']
        max_latency.append(maxlatency)

        nintynine = data[run]['global-stats']['e2e-upstream-latency'][0]['99%']
        ninty_nine.append(nintynine)

        meanlatency = data[run]['global-stats']['e2e-upstream-latency'][0]['mean']
        mean_latencies.append(meanlatency)

    json_file.close()
dropsAtSink = str(dropsAtSink).replace('[','').replace(']','')
dropsAtOne = str(dropsAtOne).replace('[','').replace(']','')
dropsAtTwo = str(dropsAtTwo).replace('[','').replace(']','')
dropsAtThree = str(dropsAtThree).replace('[','').replace(']','')
dropsAtFour = str(dropsAtFour).replace('[','').replace(']','')
dropsAtFive = str(dropsAtFive).replace('[','').replace(']','')
dropsAtSix = str(dropsAtSix).replace('[','').replace(']','')
dropsAtSeven = str(dropsAtSeven).replace('[','').replace(']','')
```

```
battery_1 = str(battery_1).replace('[','').replace(']','')
battery_2 = str(battery_2).replace('[','').replace(']','')
battery_3 = str(battery_3).replace('[','').replace(']','')
battery_4 = str(battery_4).replace('[','').replace(']','')
battery_5 = str(battery_5).replace('[','').replace(']','')
battery_6 = str(battery_6).replace('[','').replace(']','')
battery_7 = str(battery_7).replace('[','').replace(']','')
lowestlife = str(lowestlife).replace('[','').replace(']','')

appsent = str(appsent).replace('[','').replace(']','')
applost = str(applost).replace('[','').replace(']','')
appreceived = str(appreceived).replace('[','').replace(']','')
ratio = str(ratio).replace('[','').replace(']','')

min_latency = str(min_latency).replace('[','').replace(']','')
max_latency = str(max_latency).replace('[','').replace(']','')
ninty_nine = str(ninty_nine).replace('[','').replace(']','')
mean_latencies = str(mean_latencies).replace('[','').replace(']','')

with open('C:/Users/Fredrik/Dropbox/sims/results.csv', 'w') as csvfile:
    filewriter = csv.writer(csvfile, delimiter=',', quoting=csv.QUOTE_MINIMAL)
    filewriter.writerow([battery_1])
    filewriter.writerow([battery_2])
    filewriter.writerow([battery_3])
    filewriter.writerow([battery_4])
    filewriter.writerow([battery_5])
    filewriter.writerow([battery_6])
    filewriter.writerow([battery_7])
    filewriter.writerow([lowestlife])

    filewriter.writerow([appreceived])
    filewriter.writerow([appsent])
    filewriter.writerow([applost])
    filewriter.writerow([ratio])

    filewriter.writerow([max_latency])
    filewriter.writerow([ninty_nine])
    filewriter.writerow([mean_latencies])
    filewriter.writerow([min_latency])

    filewriter.writerow([dropsAtSink])
    filewriter.writerow([dropsAtOne])
    filewriter.writerow([dropsAtTwo])
    filewriter.writerow([dropsAtThree])
    filewriter.writerow([dropsAtFour])
    filewriter.writerow([dropsAtFive])
    filewriter.writerow([dropsAtSix])
    filewriter.writerow([dropsAtSeven])
csvfile.close
```