Message Passing Interface for Massive Simulation of Mobile Ad-hoc Networks

Master Thesis - 10th Semester Software Engineering

ds101f19

Cassiopeia
Department of Computer Science
Aalborg University

Written in LaTeXfrom a template made by Jesper Kjær Nielsen.

Software Engineering
Aalborg University
http://www.cs.aau.dk/

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Message Passing Interface for Massive Simulations of Mobile Ad-hoc Networks

Theme:
Distributed Systems

Project Period:
Spring Semester 2019

Project Group:
ds101f19

Participant(s):
Charlie Dittfeld Byrdam
Jonas Kloster Jacobsen

Supervisor(s):
Jiri Srba
Peter Gjøl Jensen

Copies: 3

Page Numbers: 56

Date of Completion:
June 7, 2019

Abstract:

A Mobile Ad-hoc Network (MANET) is a decentralised wireless network where nodes communicate directly with each other using radio and require Medium Access Control (MAC) protocols to provide energy efficient communication. This aim of this project is to simulate the MAC protocols and to provide an alternative to real-life testing. The goal is to be able to perform repeatable experiments in a controlled topology environment. We propose a C++ library for writing and running, simulations of MANETs, using MPI. With this library, it is possible to write C++ implementations of communication and MAC protocols, for MANETs, such as ALOHA or LMAC, and perform repeatable experiments, where our library emulate the physical radio hardware and simulate radio communication between the emulated hardware. We propose a method for modelling link path loss using building footprints between nodes, on OpenStreetMap map tiles, modelling using real-life field measurements. Our experiments show that we can simulate 100 nodes in about 45 minutes, while using 128 cores, and that simulation time scales significantly with an increasing number of nodes.

Aalborg University, June, 2019

Charlie Dittfeld Byrdam
<cbyrda14@student.aau.dk>

Jonas Kloster Jacobsen
<jkja14@student.aau.dk>

Summary

A Mobile Ad-hoc Network (MANET) is a decentralised wireless network where nodes communicate directly with each other using radio, and require MAC protocols to provide energy efficient communication. The aim of this project is to simulate the MAC protocols, and to provide an alternative to real-life testing. The goal is to be able to perform repeatable experiments in a controlled topology environment.

Our project proposes a Message Passing Interface (MPI) C++ library for writing, and running, simulations of the network protocol behind the mesh communication in a MANET, modelling link path loss to simulate packet loss and collisions caused by interfering transmitters, where the physical devices, and the communication between these, are emulated entirely using software. With our library, it is be possible to write a C++ implementation of communication protocols, such as Lightweight Medium Access Protocol (LMAC) [13] or Slotted ALOHA [10], using a simple interface header file resembling a traditional hardware interface, and perform simulations with these, where each physical device is emulated by different CPUs on the MCC compute cluster at AAU [12].

The primary contribution of this thesis is the Coordinator that facilitate wireless communication between the emulated physical devices. The Coordinator allow us to simulate wireless communication in virtual time, where the Coordinator is able to skip periods of inactivity, reducing the time required to run real-time simulations.

We present arguments of correctness for the Coordinator and the hardware functions, analysing each of the different possible cases for when the Coordinator processes an action, starting with a concrete example, and finishing with a generalisation of each of the scenarios.

Additionally, we propose a method for modelling and computing link path loss by using building footprints between wireless radio transmitters and receivers, with map tiles from OpenStreetMap obtained with the Mapbox Maps Service API, and show how our method compares with another link modelling method, that uses angles between two wireless links, that share a common transmitter or receiver, to model the correlation between them.

We have implemented the LMAC [13] protocol using our C++ library, to show how our C++ library can be used to simulate the protocol, and we present the results of our scalability experiments for the Coordinator and the interface between the protocol and the Coordinator. We do, however, face significant scalability problems, as we rely on a single centralised Coordinator. Our experiments show that we can simulate 100 nodes in about 45 minutes, while using 128 cores, and that simulation time scales significantly with an increasing number of nodes.

Finally, we propose extensions to a network topology visualisation tool created by Peter Gjøl Jensen, in which we are now able to visualise communication and protocol logs, that are generated by running simulations using the Coordinator and our library.

Preface

This thesis documents the project work of a 10th semester group of Software Engineering Masters students at Aalborg University.

We would like to thank Jiri Srba and Peter Gjøl Jensen for their excellent supervision throughout this multi-semester project. Additionally, we would like to thank Rasmus Liborius Bruun for his help with link modelling, as well as supplying us with data from the Reachi project, and Madalina-Cristina Bucur for her help with her random-walk model.

Contents

Chapter 1

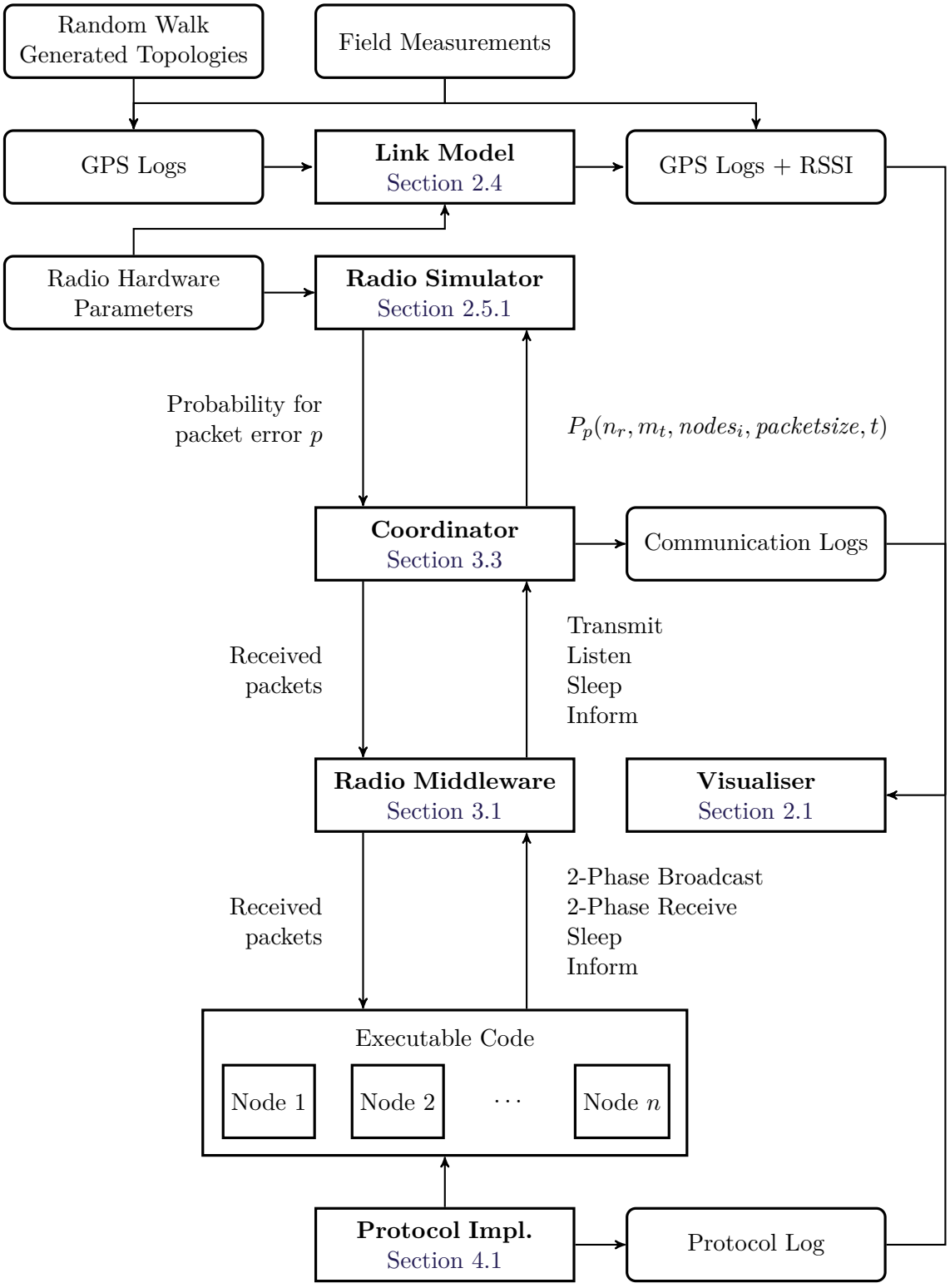
Introduction

A MANET is a decentralised wireless network that requires no pre-existing infrastructure, such as routers or access points [9]. Instead, each node in the ad-hoc network are battery powered and communicates directly with each other using a radio. Due to this, MANETs rely on wireless networking protocols to provide energy efficient communication in the network. Because of the ad-hoc nature of MANETs, common applications consist of enabling communication in emergencies such as natural disasters, or for military conflicts.

The goal of this thesis is to simulate wireless network protocols for communication in a mobile setting. Ideally, the capabilities of a wireless network protocol are tested in a real-life scenario, using physical devices for radio communication. This poses interesting challenges such as scalability, and repeatability. Scaling a real-life test requires a significant amount of effort and investment of both money and time, and repeating the same test over and over becomes near impossible, the larger the scale. Our goal is to be able to perform repeatable experiments in a controlled topology environment.

Our thesis proposes a MPI C++ library for writing, and running, simulations of the network protocol behind the mesh communication in a MANET, modelling link path loss to simulate packet loss and collisions caused by interfering transmitters, where the physical devices, and the communication between these, are emulated entirely using software. With our library, it is be possible to write a C++ implementation of communication protocols, such as LMAC [13] or Slotted ALOHA [10], using a simple interface header file resembling a traditional hardware interface, and perform simulations with these, where each physical device is emulated by different CPUs on the MCC compute cluster at AAU [12].

The primary contribution of this thesis is the Coordinator that facilitate wireless communication between the emulated physical devices. The Coordinator allows us to simulate wireless communication in virtual time, where the Coordinator can skip periods of inactivity, reducing the time required to run real-time simulations. In addition to this, we propose a method for modelling and computing link path loss using building footprints between nodes of a link, on OpenStreetMap map tiles obtained via the Mapbox Maps Service API [4]. Finally, we propose extensions to a network topology visualisation tool created by Peter Gjøl Jensen, in which we are now able to visualise communication and protocol logs, generated by running simulations with our library.


Random Walk Generated Topologies
Field Measurements
GPS Logs
Link Model
Section 2.4
GPS Logs + RSSI
Radio Hardware Parameters
Radio Simulator
Section 2.5.1
Probability for packet error p
$P_p(n_r, m_t, nodes_i, packetsize, t)$
Coordinator
Section 3.3
Communication Logs
Received packets
Transmit
Listen
Sleep
Inform
Radio Middleware
Section 3.1
Visualiser
Section 2.1
Received packets
2-Phase Broadcast
2-Phase Receive
Sleep
Inform
Executable Code
Node 1
Node 2
· · ·
Node n
Protocol Impl.
Section 4.1
Protocol Log

1.1 Related Work

In "Modeling and Efficient Verification of Broadcasting Actors" [14] the authors present an extension to the actor-based modelling language Rebeca [11], that enable broadcast communication between actors (nodes), to allow modelling of MANETs. The authors provide a framework to model MANETs for a static topology, with no support for mobility. The same authors further extend Rebeca to add key features of wireless ad hoc networking, such as mobility (dynamic topologies), local broadcasting within a transmission range, and energy consumption in "Modeling and efficient verification of wireless ad hoc networks" [15]. The modelling language lacks features such as lossy transmissions (packet loss) and non-deterministic behaviour, and generally abstracts away from wireless communication, to focus on modelling and verification of MANET protocols. Our approach is generally more un-restrictive, as we allow protocol implementations in C++, whereas the paper is restricted to a fixed formalism. One of the major advantages of using Rebeca is that the modelling language makes it possible model check and verify MANET protocol implementations, and explore the full state space for the model of a protocol.

The paper "Modeling and Evaluation of Wireless Sensor Network Protocols by Stochastic Timed Automata" [16] proposes a method to analyse and evaluate Wireless Sensor Network (WSN) protocols using Stochastic Timed Automata, along with the non-deterministic behaviour of WSNs, such as lossy transmission and dynamic topologies. The authors utilise statistical model checking to evaluate the performance of WSN protocols, as well as checking the correctness of the protocols. This approach is somewhat similar to the approach in [14] and [15], but the protocols evaluated are instead modelled using the UPPAAL model checker and does include non-deterministic behaviour. They show in the paper that their method can model and evaluate network topologies of up to 100 nodes by using statistical model checking in UPPAAL.

In the paper "Simulating MANETS: A Study using Satellites with AODV and AntHocNet" [7], the authors present a network simulator for satellite networks called SatSim. The authors argue that a satellite constellation can be thought of as an extreme example of a MANET. SatSim includes features such as a bit error rate, determined for each packet using a representative link budget, where packets are randomly dropped if the bit error rate exceeds a specific threshold. This approach is similar to ours in that we also use a link budget (the link path loss model) to compute the probability for packet errors. Our approach differs, in that we expand upon this by also simulating collisions, caused by interfering transmitters.

Chapter 2

Radio Physics

The goal of this chapter is to be able to estimate the probability, based on the topology of a wireless network, of how likely it is that packet loss will happen during transmission between two nodes.

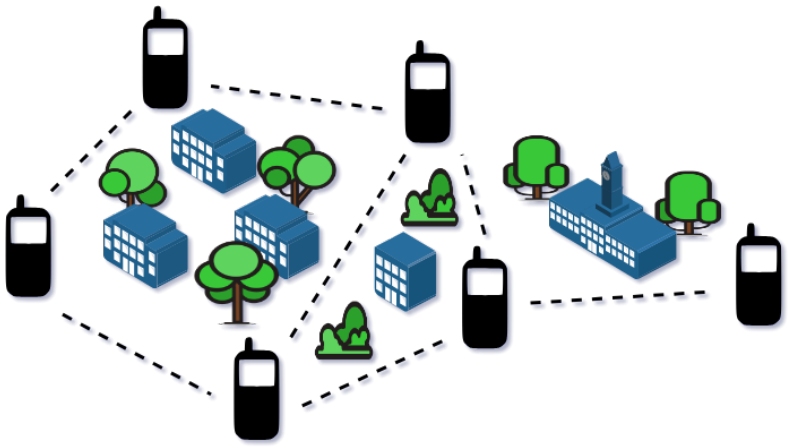


Figure 1: A wireless network topology.

Figure 1 shows a sample wireless network topology for a Mobile Ad-hoc Network (MANET). The network consists of mobile devices (nodes), and the communication between these (links). Nodes are "linked" with other nodes when they can communicate wirelessly. Wireless communication relies on the transmission and reception of electromagnetic waves [1, p. 10] (radio signals), and the strength, or quality, of a wireless link, is described by the signal loss occurring when propagating the signal from transmitter to receiver, and is measured by the Received Signal Strength Indication (RSSI) (a negative value, where a value close to 0 is better). In [1], the term path loss is used to describe this signal loss and is determined in part by the physical distance between transmitter and receiver, but also by physical objects and terrain, like buildings or forests. A major consideration for mobile networks is that the topology is dynamic. Nodes move around, causing links to disappear, or new links to appear, thus changing the topology of the network. Section 2.1 introduces the Visualiser and presents the proposed extensions. Section 2.2 introduces a series radio specific terms we use throughout the thesis. Section 2.3 presents the link path loss model from [1], and Section 2.4 introduces an alternative path loss model based on building footprints between nodes in a links, by using OpenStreetMap map tiles. Finally, Section 2.5 presents the method for simulating packet loss and transmission interference.

2.1 Visualiser Tool

The Visualiser is a tool written in Python and JavaScript, created by Peter Gjøl Jensen. The tool was created to aid in the visualisation of MANET topologies and works by importing a log file with GPS coordinates and timestamps for a series of nodes. Using the tool, it is possible to visualise the position and movement for all nodes in a network. A snippet of a GPS log can be seen below. Each line consists of the identifier of the node, the latitude and longitude coordinates for the node, and the timestamp for the coordinates in milliseconds. We found the tool to be able to handle visualisations up to around 100 nodes.

```
#id,lat,lon,timestamp
64,14.629879,121.096137,158980000.000000
64,14.629874,121.096132,159000000.000000
64,14.629878,121.096128,159020000.000000
64,14.629890,121.096143,159040000.000000
64,14.629892,121.096142,159060000.000000
64,14.629896,121.096141,159080000.000000
64,14.629893,121.096164,159100000.000000
64,14.629947,121.096083,159120000.000000
64,14.630107,121.095976,159140000.000000
64,14.630283,121.095885,159160000.000000
64,14.630525,121.095786,159180000.000000
```

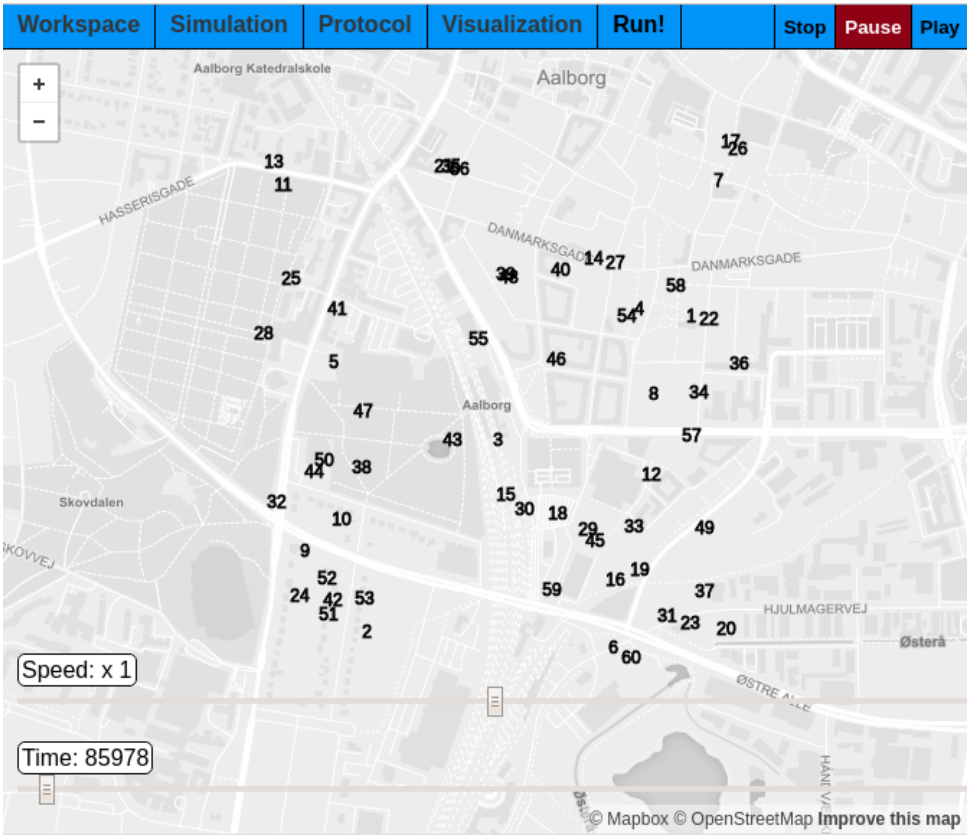


Figure 2: A visualised GPS log.

Figure 2 shows a screen-shot from the Visualiser, with a GPS log loaded. When a GPS log is

loaded, the Visualiser can be started by pressing the “Play” button, or the “space” key. The speed of the visualisation can be controlled with the “Speed” slider in the bottom, and the current time of the visualisation can be controlled with the “Time” slider.

2.1.1 Extensions

We propose three extensions to the visualiser tool. The first is to visualise the link between nodes using an annotated version of the GPS log, where each line is annotated with the RSSI of a link between nodes in the log, as shown below. In the annotated log, each link for a given node, to another node, is annotated, after the timestamp, with the identifier of the other node, and the RSSI between them. Figure 3 shows an example of a very connected network where links between nodes are visualised by a colour gradient, where a link with a yellow colour has a better RSSI than a link with a red colour. It is possible to print the RSSI value for the links as well.

```
#id,lat,lon,timestamp,id1,rssi1,id2,rssi2,id3,rssi3, ...
65,14.630107,121.096749,157820000.000000,67,-56,69,-70,71,-13, ...
65,14.630129,121.096905,157840000.000000,67,-58,69,-61,73,-65, ...
65,14.630189,121.097116,157860000.000000,67,-55,69,-54,73,-71, ...
65,14.630318,121.097294,157880000.000000,67,-65,69,-66,71,-13, ...
65,14.630330,121.097545,157900000.000000,67,-79,69,-48,73,-79, ...
65,14.630358,121.097725,157920000.000000,67,-85,69,-66,71,-28, ...
65,14.630243,121.097900,157940000.000000,69,-84,71,-35,83,-67, ...
65,14.630082,121.098037,157960000.000000,71,-45,83,-70,89,-43, ...
65,14.629960,121.098165,157980000.000000,71,-20,83,-75,89,-38, ...
65,14.629729,121.098192,158000000.000000,83,-81,89,-42,97,-80, ...
```

The second extension is to be able to replay the communication between nodes when simulating a protocol. This log is generated by the Coordinator (introduced in Section 3.3), and a line is added whenever a packet is either dropped or received during transmission. Each line states whether the packet was received or dropped, the identifier of the transmitter and receiver, the number of bytes sent, the RSSI for the transmission, the probability for packet error, that decided whether the packet was dropped or not, the interfering power and the number of interfering transmitters (if any), and finally, the start and end time of the transmission. With this log, it is possible to visualise any transmissions by drawing a unidirectional arrow from the transmitter to the receiver, within the time interval of the transmission.

```
#received,tx_id,rx_id,bytes,rssi,pep,int_power,ints,tx_start,tx_end
recv,1,2,2,-102.419,5.74662e-05,0,0,2233,2692
recv,1,5,2,-102.419,5.7577e-05,0,0,2233,2692
drop,1,6,2,-110.697,0.494838,0,0,2233,2692
drop,1,6,24,-110.697,0.999724,0,0,12692,18209
recv,1,5,24,-102.419,0.000690706,0,0,12692,18209
recv,1,2,24,-102.419,0.000689376,0,0,12692,18209
recv,1,2,2,-102.419,5.74662e-05,0,0,32002048,32002507
recv,1,5,24,-102.419,0.000690706,0,0,32012507,32018024
recv,1,2,24,-102.419,0.000689376,0,0,32012507,32018024
```

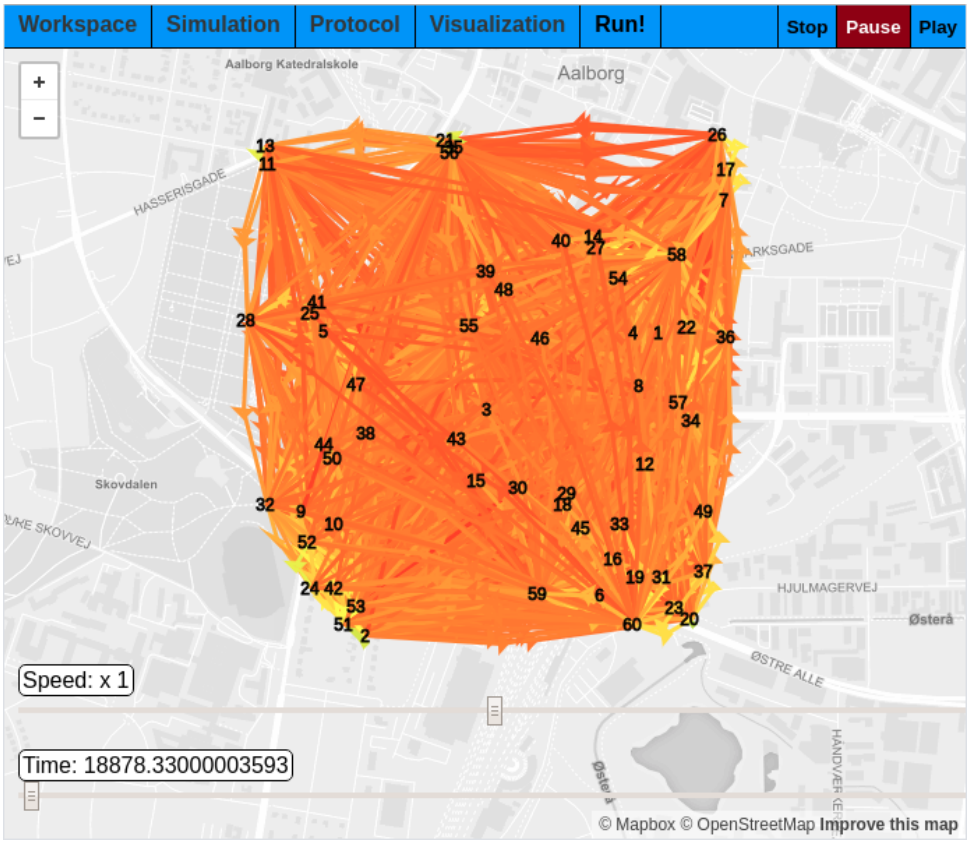


Figure 3: A visualised GPS log with coloured links, based on RSSI.

The third, and final, extension is to replay state changes of a protocol. When simulating a protocol like the LMAC protocol (introduced in Section 4.1), where each node proceeds through a number of states, we log each of the state changes, as shown below, and can replay these state changes in the Visualiser.

```
#timestamp,id,state
0,10,i
0,1,i
0,9,i
800.009,1,0
800.002,2,w
5600,2,d
800.002,5,w
4800,5,d
5600.01,5,5
6400.01,2,6
6400,6,w
8800,6,d
```

A visualised example of the second and third extension for an execution of the LMAC protocol can be found in Figure 20 in Section 4.1, and in Figure 4.

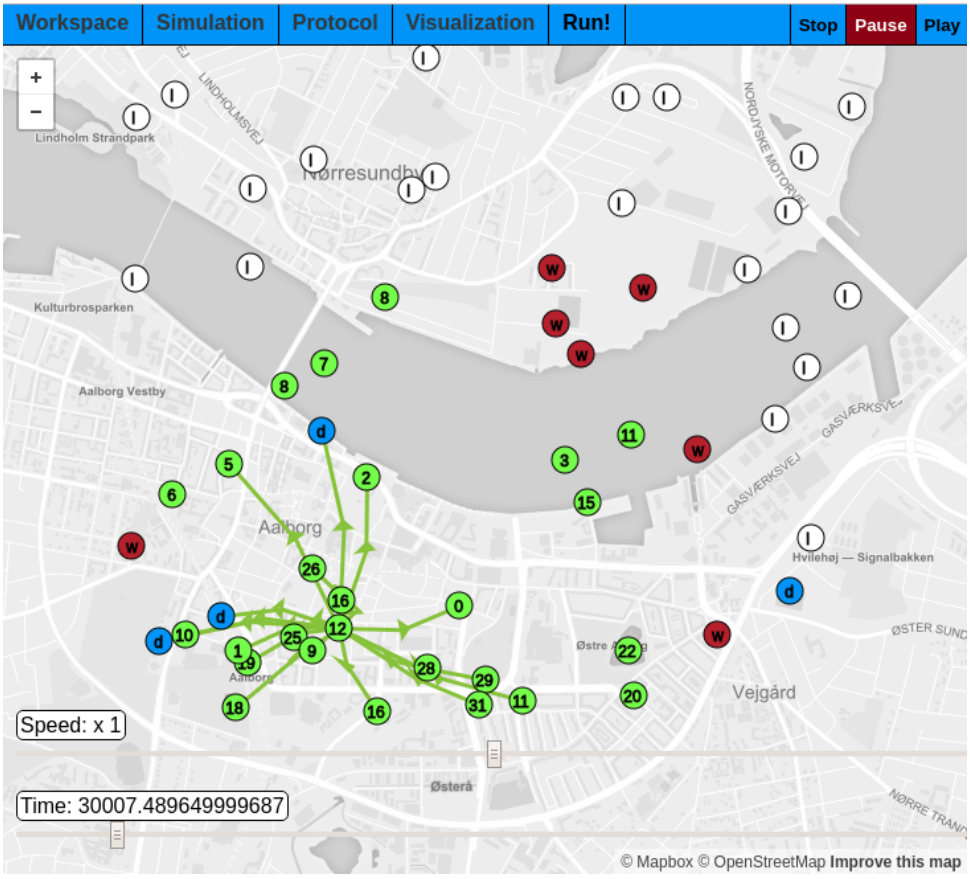


Figure 4: Communication and protocol logs visualised.

The complete source code for the Visualiser tool can be found on GitHub:
`https://github.com/Joklost/manet-simulations/tree/master/tools/visualiser`

2.2 Radio Hardware

In this section, we introduce a series of terms, such as RSSI, transmission power, dBm, and path loss, that we utilise throughout the thesis.

- dBm is a logarithmic scale, measuring power of a wireless radio signal [8].
- RSSI is the perceived signal strength of a link, in dBm [8].
- path loss is the signal loss inflicted by the propagation of a radio signal from transmitter to receiver [1, p. 10].
- Transmission power is the actual amount of power a transmitter uses to transmit packets, in dBm. RSSI can be computed by subtracting path loss from the transmission power [1]. For the Reachi devices, the transmission power is 26 dBm [1].
- The baud rate is the rate at which information can be transferred as a wireless signal [6], and is equivalent to bits per second. For the Reachi devices, the baud rate is 34800 Hz [1].

Equation 2.1 computes the amount of time required to transmit a packet, in microseconds, based on the baud rate f_s and the size of a packet in bytes. For a packet where $|packet| = 20$

bytes would take $transmissiontime(|packet|) = 4597$ microseconds, with baud rate $f_s = 34800$ Hz.

$$transmissiontime(packetsize) = \frac{1000000}{f_s} \cdot (packetsize \cdot 8) \tag{2.1}$$

2.3 Link Path Loss

In this section, we present the method for simulating link path loss from [1], as well as why the model does not work for our needs. For our simulations, we want to simulate the performance of nodes in a MANET. The performance is, however, heavily dependent on network conditions and the capabilities of the technology [1, p. 10]. The author of [1] presents methods for evaluating the performance of a wireless network and proceeds to introduce methods for simulating path loss on a multi-link model, based on real-world performance measurements.

The author of [1] describes the path loss of a link to be the sum of two parts: A deterministic distance-dependent part, that describes the mean signal attenuation at any given link distance, and a stochastic shadow fading part, which is the path loss caused by terrain, buildings, vegetation and cars. With this path loss, it is possible to simulate the RSSI on a given link, by subtracting the path loss from the transmission power of the simulated radio.

$$pl_d(l) = 55 \log_{10}(d(l)) - 18.8 \tag{2.2}$$

The distance-dependent path loss is computed using the $pl_d(l)$ function shown in Equation 2.2 [1, p. 25], where the function $d(l)$ denotes the distance of a link in meters. Computing the shadow fading path loss, on the other hand, is not as trivial. The shadow fading part of the path loss is based on the correlation between angles of link pairs sharing a common nodes, and bears a significant practical limitation in the sense that the shadow fading part depends on a Cholesky factorisation with a computational complexity of $O(N^6)$ [1, p. 31], where N is the total number of nodes in the network.

Through personal communication with the author of [1], we received access to logs from field experiments for the Reachi project. These field experiments were conducted in different locations with an early prototype of the Reachi device. The logs contain Global Positioning System (GPS) coordinates, as well as RSSI information for detected neighbours of each node. Examining these logs have shown discrepancies between the path loss model from [1] and the measured RSSI. Figure 5 plots samples drawn from the $pl_d(l)$ function and measurements from a log containing field measurements from an experiment in Marikina, in the Phillippines. Since the log contained a total of 17761 links, the measurements are summarised based on the distance of the link, and each link was sorted into distance buckets with 20-meter intervals. The average RSSI for all links in a bucket is plotted in Figure 5. The plot shows that the RSSI computed with the distance-dependent path loss does not fit with the measured RSSI.

As mentioned earlier, the shadow fading path loss is based on the correlation between angles of link pairs that share a common node. An assumption for this is that link pairs with a high correlation, where the angle between them is low, will have close to the same shadow fading [1]. However, this does not seem to be the case. Figure 6 shows a plot where we compare the Marikina log from earlier, with another field experiment log from Rude Skov. For both logs, pairs of links sharing a common node were sorted, based on the angle between them, into buckets of 5 °intervals, and we computed the average RSSI for these buckets, after removing the distance-dependent path loss. This means that only the shadow fading part of the path loss is included in the RSSI plotted in Figure 6. Under the assumption that highly correlated link pairs should result in less shadow fading path loss, the traces on Figure 6 should increase gradually as the angle increases. This is not the case.

Because of this, and the fact that computing the shadow fading path loss is not feasible for a very large number of nodes, we instead propose our model for approximating the shadow fading path loss.

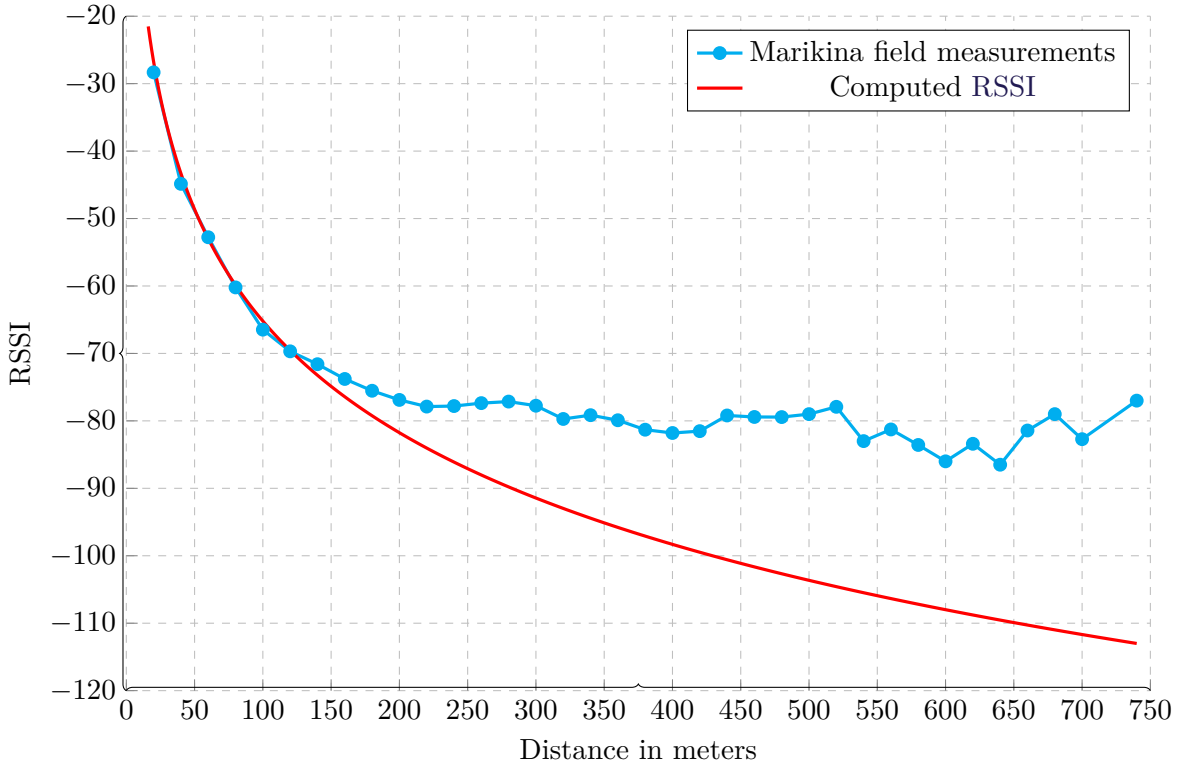


Figure 5: Average RSSI pr. distance bucket.

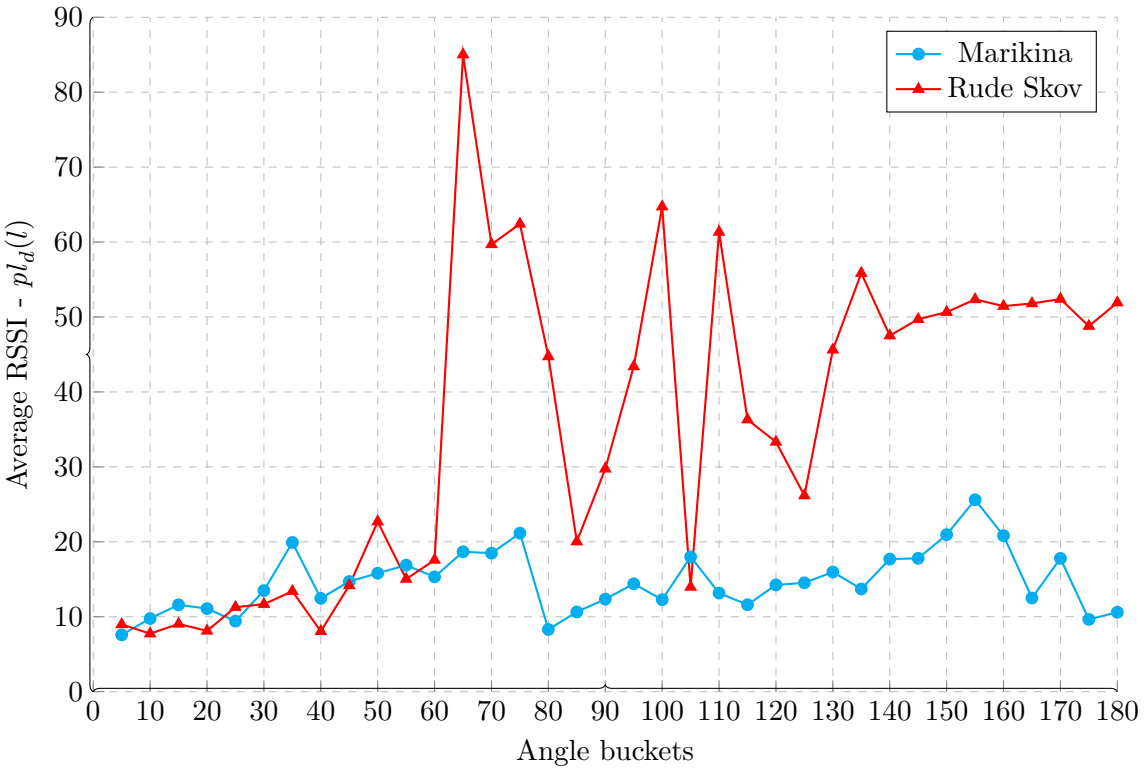


Figure 6: Average RSSI per angle bucket without distance dependent path loss.

2.4 Link Modelling

In this section, we propose a method for modelling and computing link path loss using building footprints between nodes of a link, on OpenStreetMap map tiles obtained through the Mapbox Maps Service API [4]. The model computes the path loss based on the distance of the link and the percentage of that distance that is in a building. Buildings and other environmental obstructions, which is part of the shadow fading path loss from [1], should cause a higher path loss, as it is harder for the radio signal to propagate through buildings.

The main idea is to generate a map of the area that contains nodes of a link as an image, and when computing the path loss of that link, count the percentage of all pixels in a straight line between the nodes in a link that are considered buildings on the map. The pseudo code description of this can be seen in Algorithm 1.
To compute the total path loss, we first define two functions: $cvpl(l)$ (Clear View Path Loss (CVPL)) and $bopl(l)$ (Building Obstructed Path Loss (BOPL)). Both functions compute the distance-dependent path loss, similarly to the pl_d function from Section 2.3, but rather than computing the total distance based path loss, we need to define a similar function for the distance with a clear view, and a similar function to the distance where buildings obstruct the signal. To do this, we define this as an optimisation problem, where we want to find the optimal constants for the two functions, by minimising the difference between the computed RSSI and the measured RSSI for a set of links L. The $compRSSI(l) = tx_{power} - (\alpha \cdot (\ln(d(l))/\ln(\delta)) + \beta)$ function denotes the computed RSSI for a link with the chosen values for α, β, and δ.

The problem is defined as follows:

- Input: A set of links L.

Algorithm 1: The CompBuildingPct function.

Input: $(x_1, y_1), (x_2, y_2)$
Output: Percentage building between points.

```
Function CompBuildingPct((x1, y1), (x2, y2))
    pixels ← 0
    buildings ← 0
    while λ ∈ {0 . . . 1} do
        (x, y) ← λ · (x1, y1) + (1 − λ) · (x2, y2)
        if position (x, y) is a building then
            buildings ← buildings + 1
        pixels ← pixels + 1
    return buildings / pixels
```

- Output: Optimal values for α, β, δ.
- Goal: Minimise the $score(\alpha, \beta, \delta)$ function:
  $score(\alpha, \beta, \delta) = \sum_{l \in L} (compRSSI(l) - measuredRSSI(l))^2$

2.4.1 Greedy Approach

To solve the optimisation problem, we have chosen a greedy approach. First, to compute the optimal values for the $cvpl(l)$ function, we compile a set of links L, where the computed building percentage is below 5 %, and for the $bopl(l)$ function, we compile a set of links L where the computed building percentage is above 80 %. Ideally, we would like for the building percentage to be close to 100 %, but the number of links in the Marikina log with more than 95 % of buildings is very low. With these sets, we attempted to find the optimal values for α, β, and δ by going through $\alpha, \beta \in \{-100, \ldots, 100\}$ with increments of 0.5, and $\delta \in \{2, \ldots, 100\}$ with increments of 1. This resulted in the following values for the $cvpl(l)$ and $bopl(l)$ functions:

$$cvpl(l) = 48.5 \cdot (\ln(d(l)) / \ln(77)) + 37.5 \tag{2.3}$$

$$bopl(l) = 67 \cdot (\ln(d(l)) / \ln(57)) + 11.5 \tag{2.4}$$

With these two functions defined, we can compute the total path loss for a link. The function $p(l)$ denotes the points for the links, as required by the input to the CompBuildingPct function.

$$pl(l) = (cvpl(l) \cdot (1 - \text{CompBuildingPct}(p(l)))) + (bopl(l) \cdot \text{CompBuildingPct}(p(l))) \tag{2.5}$$

Finally, with the $pl(l)$ function, we can compute the RSSI for the link l:

$$RSSI_{dBm}(l) = tx_{power} - pl(l) \tag{2.6}$$

2.4.2 Evaluation

The functions $cvpl(l)$ and $bopl(l)$ have been plotted on Figure 7. *bopl* do result in greater path loss, however, the plot also reveals that up to 100 meters, $bopl(l)$ compute a better RSSI compared to $cvpl(l)$. To further examine this, each function has been plotted with their training set. $cvpl(l)$ on Figure 8 and $bopl(l)$ can be seen in Figure 9.

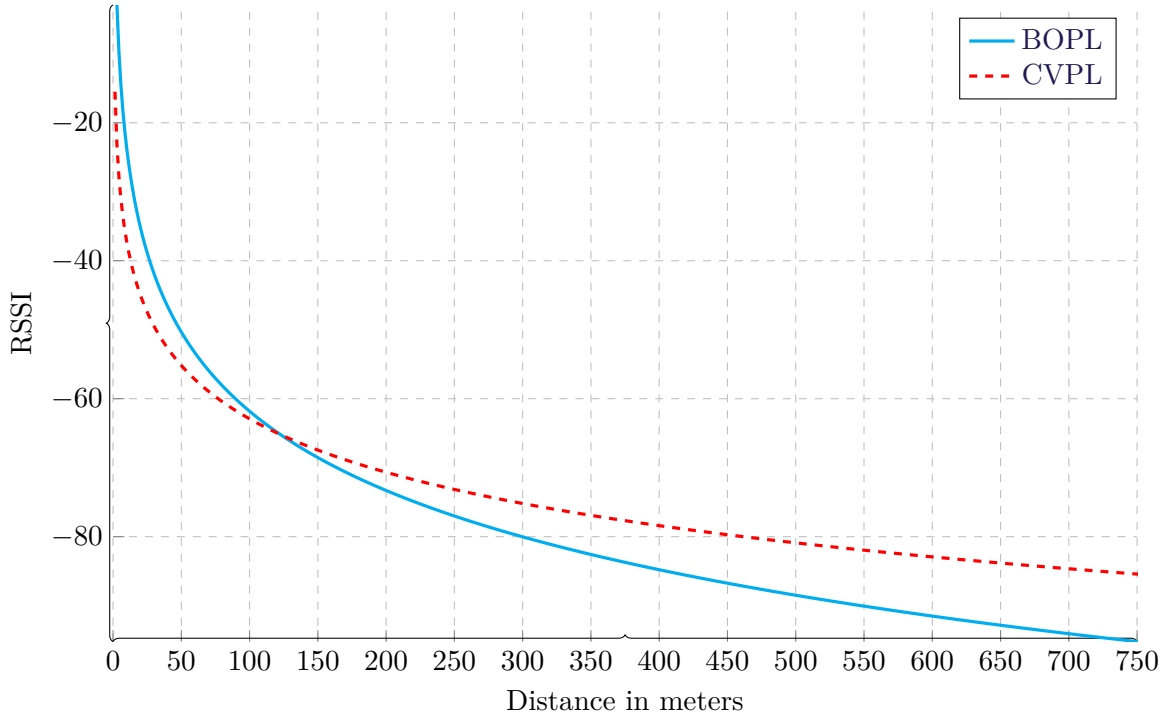


Figure 7: Plot showing samples drawn from CVPL and BOPL

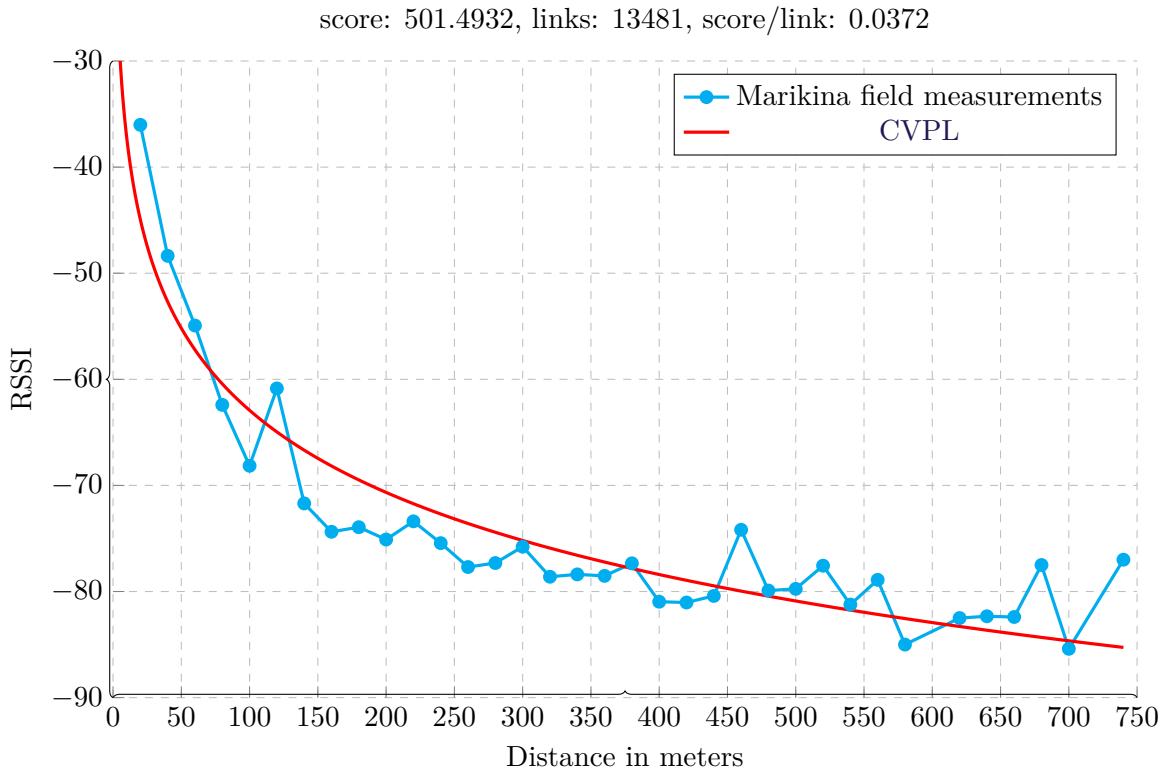


Figure 8: Field measurements with building percentage below 5 %.

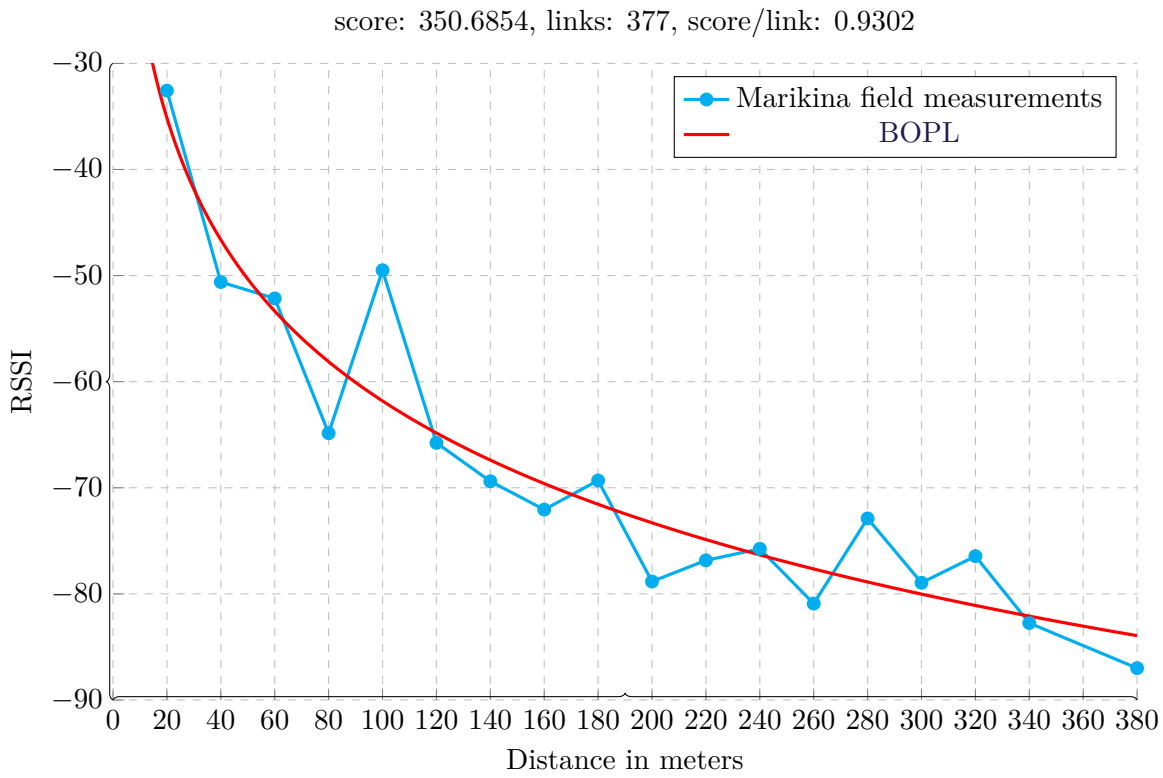


Figure 9: Field measurements with building percentage above 80 %.

With the optimal values for the $cvpl(l)$ function, the final score was 501.4932 for 13481 links with less than 5 % buildings. Figure 8 shows how well this function fits with the measured values.

For the $bopl(l)$ function, we got a final score of 350.6854 for 377 links with more than 80 % buildings. This is a very large difference compared to the 13481 links for the $cvpl(l)$, which means that we might experience lower precision for the $bopl(l)$ function.
Finally, a comparison of the computed RSSI values from Equation 2.6 with the measurements from the Marikina log is shown in Figure 11. The plot shows that the function is slightly off on the from about 75 meters to 300 meters, but compared to Figure 5, we do see an improvement.

(a) Field measurements visualised.

(b) Computed RSSI values visualised.

Figure 10: Marikina field measurements and computed RSSI values.

Figure 10 contains YouTube links to two visualisations, where Figure 10a visualises the Marikina field measurements, and Figure 10b visualises the computed RSSI values. Both visualisations highlight the same subset of the links, to make it easier to follow.

The complete source code for the C++ implementation can be found on GitHub:
`https://github.com/Joklost/sims2`

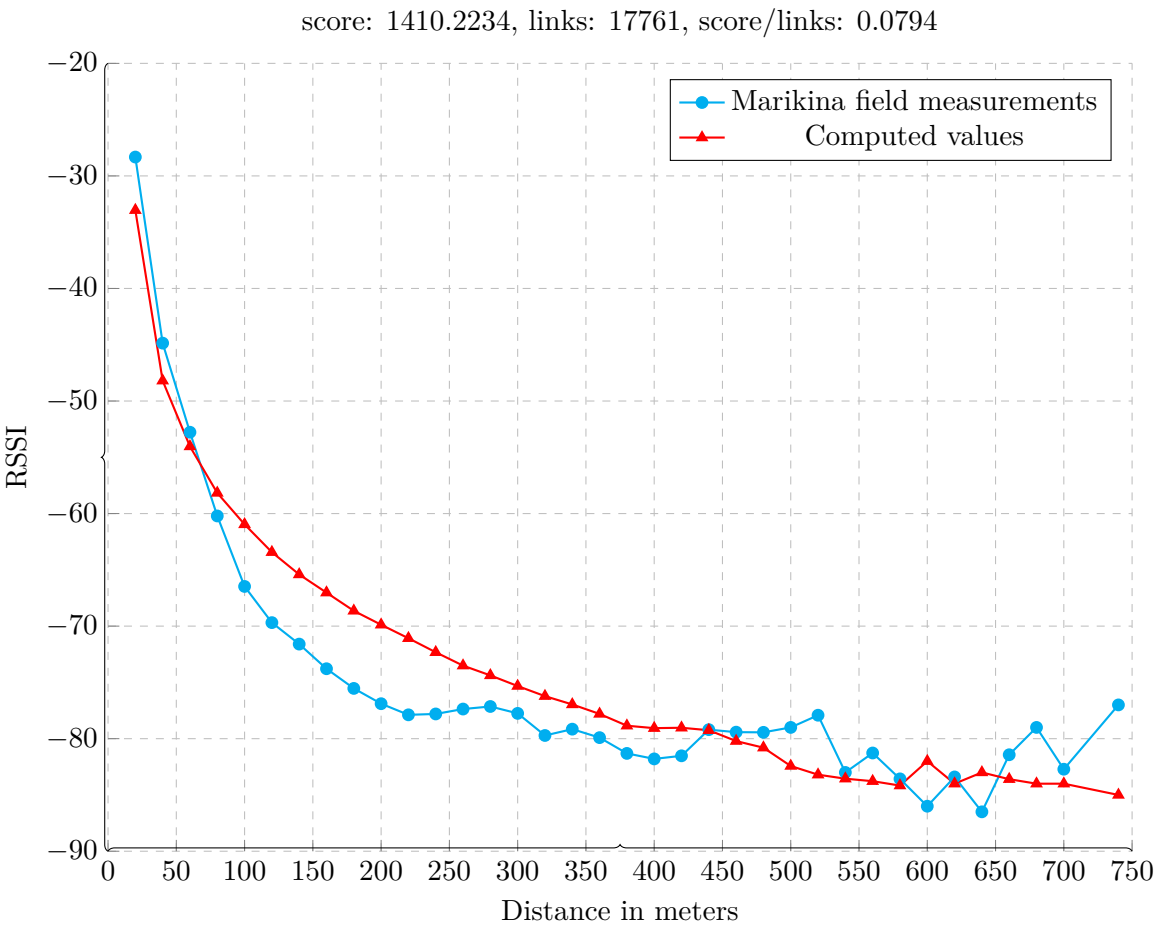


Figure 11: Field measurements vs. computed values.

2.5 Radio Simulation

To simulate loss of packets during radio communication, we introduce the packet error probability. The packet error probability is the probability for any form of error occurring during the transmission and reception of a packet through wireless radio communication. The probability for packet error is calculated using the RSSI on a link between two nodes, the size of the packet, as well as the Signal to Noise Ratio (SNR), including interference from nearby transmitting nodes. The computations in this section are derived from [5], as well as personal communication with the author of [1].

2.5.1 Probability for Packet Error

The first step for computing the probability for packet error is to compute the level of background noise affecting the wireless communication, the noise power $P_{N,dB}$. This noise is calculated with the thermal noise and noise figure $P_{N,dB} = \mathit{thermalnoise} + \mathit{noisefigure}$. For the Reachi devices, we assume that the $\mathit{thermalnoise} = -119.66$ dB and the $\mathit{noisefigure} = 4.2$ dB. Next, we need to add the noise from interfering transmissions happening at the same time. This is done by adding the sum of the RSSI from interfering transmitters to the noise power $P_{N,dB}$, giving us the noise power with interference $P_{NI,dB}$ on the specific link between a receiving node n_r and a transmitting node n_t at a given time t. The set of currently transmitting and interfering nodes are denoted by $nodes_i$ and the function $RSSI_{dBm}(n, m, t)$ denotes the RSSI, in dBm, on the link between nodes n and m at time t. We assume the RSSI on a link to be reciprocated, which means that $RSSI_{dBm}(n, m, t) = RSSI_{dBm}(m, n, t)$.

$$P_{NI,dB}(n_r, m_t, nodes_i, t) = 10 \log_{10} \left(10^{\frac{P_{N,dB}}{10}} + \sum_{m \in nodes_i} 10^{\frac{RSSI_{dBm}(n_r, m, t)}{10}} \right) \tag{2.7}$$

Note that as both the noise power $P_{N,dB}$ and the RSSI is in dB (a logarithmic scale), we first need to convert the values to a linear scale, before we can compute the sum of the background noise and the interfering noise, and then finally convert the value back into a logarithmic scale.

With the noise and interference power $P_{NI,dB}$, we can compute the Signal to Noise (and Interference) Ratio (SN(I)R), γ_{dB}. The SN(I)R compares the RSSI of the signal to the level of the background noise, as well as the noise from interfering transmitters. The ratio is computed by subtracting the noise power $P_{NI,dB}$ from the RSSI of a link.

$$\gamma_{dB}(n_r, m_t, nodes_i, t) = RSSI_{dBm}(n_r, m_t, t) - P_{NI,dB}(n_r, m_t, nodes_i, t) \tag{2.8}$$

We use the SN(I)R γ_{dB} to compute the bit error probability P_b:

$$P_b(n_r, m_t, nodes_i, t) = \frac{1}{2} erfc \left(\sqrt{\left(\frac{10^{\frac{\gamma_{dB}(n_r, m_t, nodes_i, t)}{10}}}{2} \right)} \right) \tag{2.9}$$

Finally, with the bit error probability P_b, we can compute the packet error probability P_p. The packet error probability is the probability that we experience a bit error for any of the bits in the transmitted packet. The *packetsize* parameter is in bytes.

$$P_p(n_r, m_t, nodes_i, packetsize, t) = 1 - (1 - P_b(n_r, m_t, nodes_i, t))^{packetsize \cdot 8} \tag{2.10}$$

2.5.2 Example

If we assume that a node n_2 is currently listening, and nodes n_1 and n_3 is transmitting at the same time t, what is the probability for a packet error on the link between nodes n_1 and n_2 with interference $nodes_i = \{n_3\}$? For this example we assume the RSSI for the link between n_1 and n_2 to be $RSSI_{dBm}(n_2, n_1, t) = -63.750$, the RSSI between n_2 and n_3 to be $RSSI_{dBm}(n_2, n_3, t) = -74.042$, and the size of the transmitted packet to be 20 bytes (which is the size of a header packet for the Reachi protocol). First, we compute the noise power $P_{NI,dB}$:

$$P_{NI,db}(n_2, n_1, nodes_i, t) = 10 \log_{10} \left(10^{\frac{(-119.66+4.2)}{10}} + 10^{\frac{-74.042}{10}} \right) = -74.041 \tag{2.11}$$

We subtract the noise power $P_{NI,dB}$ from the RSSI to get the SN(I)R γ_{dB}:

$$\gamma_{dB}(n_2, n_1, nodes_i, t) = -63.750 - (-74.041) = 10.291 \tag{2.12}$$

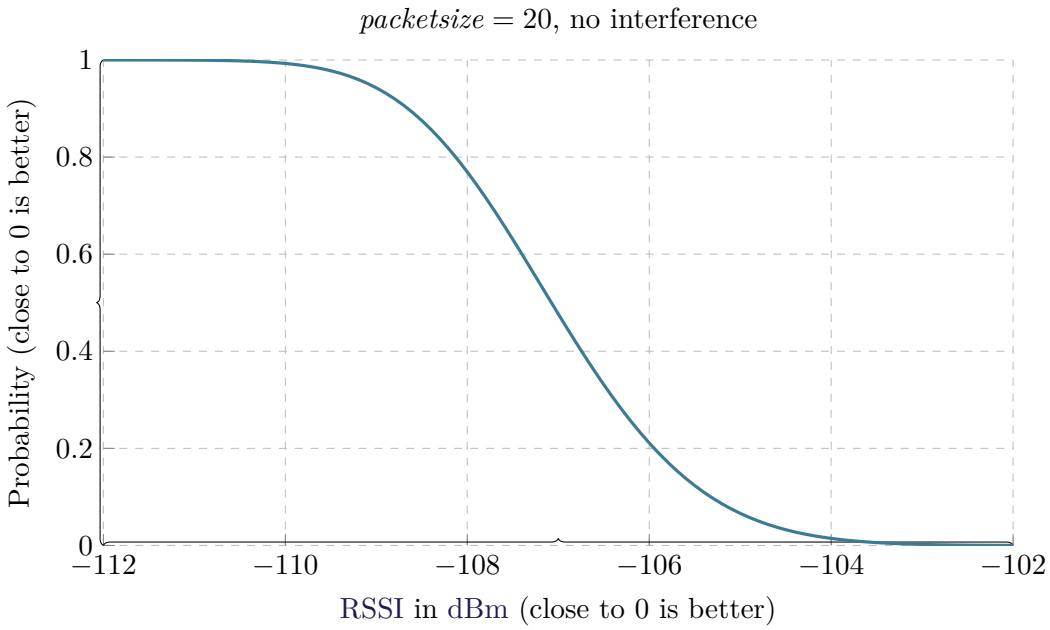
With which we can compute the bit error probability:

$$P_b(n_2, n_1, nodes_i, t) = \frac{1}{2} erfc \left(\sqrt{\left(\frac{10^{\frac{10.291}{10}}}{2} \right)} \right) = 0.000537 \tag{2.13}$$

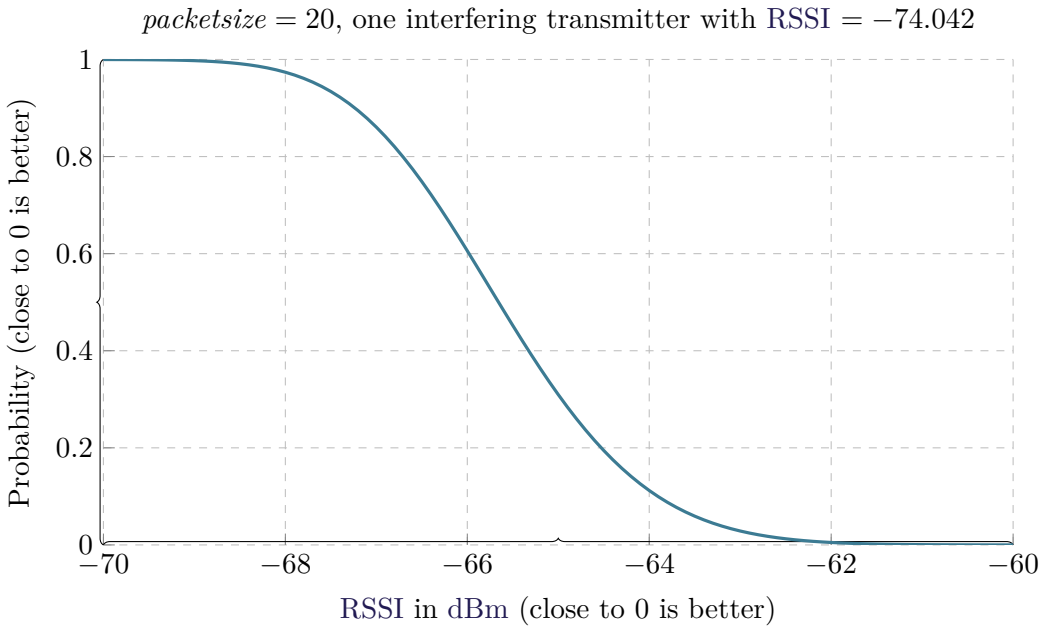
Finally, we can compute the packet error probability using the bit error probability:

$$P_p(n_2, n_1, nodes_i, t, 20) = 1 - (1 - 0.000537)^{20 \cdot 8} = 0.082 \tag{2.14}$$

This gives us an 8.2 % probability that we will experience a packet error during the transmission from n_1 to n_2 with interference from n_3, which is a significant difference in relation to the same transmission with no interfering transmitters. To demonstrate the difference, Figure 12a shows the probability for packet error with no interfering transmitters. According to the figure, an RSSI of approximately -103.0 dBm would have a probability for packet error close to zero, and an RSSI of approximately -110.0 dBm would have a probability for packet error very close to 100.0 %. Recall that for the link between n_1 and n_2 at time t, we had an RSSI of -63.750 dBm, which is significantly better than the -103.0 dBm we see in Figure 12a for a close to zero probability, but with just a single interfering transmitter, the probability for packet error increases to 8.2 %, which corresponds to what we see in Figure 12b, where an RSSI of approximately -62.0 dBm is required for a probability for packet error close to zero, with a single interfering transmitter.



(a) Probability for packet error on a link with no interfering transmitters.



(b) Probability for packet error on a link with a single interfering transmitter.

Figure 12: Probability for packet error with and without interfering transmitters.

Chapter 3

Communication

In this chapter, we introduce the Message Passing Interface (MPI) for emulating the radio functionality of wireless nodes. The interface consists of three parts: An interface for nodes to communicate, emulating radio hardware functionality, a Coordinator to coordinate the communication between nodes, and a set of low-level hardware functions for transmitting, listening, and sleeping. The hardware interface is introduced in Section 3.1, the Coordinator is presented in Section 3.3, and the hardware functions are presented in Section 3.3.2. Finally, Section 3.4 will present arguments of correctness for our interface and the Coordinator, where we show that a simulated execution, using MPI, will behave the same as a real execution.

3.1 Abstract Hardware

To simulate the physical radio hardware in our simulations, we define a hardware interface consisting of two basic hardware functions, Transmit, and Listen. The two hardware function to emulate the functionality of radio hardware. The Transmit function takes a packet of arbitrary data as input, and the node transmits the packet to nearby listening nodes. The duration of the transmission is computed based on the size of the packet, using Equation 2.1 from Section 2.2. The Listen function takes a duration as input, and the node listens for a packet, for the duration. If a packet is received, the node stops listening and returns the packet immediately. If nothing is received while listening, the function will return ***null***.

3.2 2-Phase Broadcast and Receive

As mentioned previously in Section 3.1, the time required for transmitting a packet is dependent on the size of the packet. To aid with the timing of transmitting and listening, we introduce a two-phase system. The two-phase system consists of two functions, Broadcast (Algorithm 2) and Receive (Algorithm 3).
The Broadcast function takes a *packet* of arbitrary data as input. With the packet of data, the function constructs a header packet that includes the size of the input *packet* (line 4). The size of the header packet is dependent on the design on the radio hardware. The header packet is transmitted via a call to the Transmit function, after which the function sleeps for a pre-defined *delay*. This delay allows for any listening nodes to start listening for the actual packet, after receiving the header packet, before the actual packet is being transmitted. The pre-defined delay is a global value, known by all nodes.
The Receive function takes a *duration* as input, and begins listening for the duration (line 4). The *duration* input is how long a node wants to listen for a packet. A node stops listening the

Algorithm 2: The Broadcast function.

```
1  delay ← short delay between first and second transmission
2
3  Function Broadcast(packet)
4      header ← construct header packet with |packet|
5      Transmit(header)
6      Sleep(delay)
7      Transmit(packet)
```

Algorithm 3: The Receive function.

Result: A packet or ***null***

```
1  delay ← short delay between first and second transmission
2
3  Function Receive(duration)
4      data ← Listen(duration)
5      if data ≠ header packet then
6          return null
7      packetsize ← extract size of packet from header data
8      d′ ← transmissiontime(packetsize)
9      packet ← Listen(d′ + delay · 2)
10     // The packet returned from Listen may be null
11     return packet
```

moment a packet has been received, or after the duration if nothing was received. The Listen function either returns a packet of data if anything was received or ***null***. The Receive function checks if the received data is a header packet (line 5), and returns ***null*** if the received packet was not a header packet. Otherwise, the size of the incoming packet is extracted from the header packet, and the expected transmission time is calculated using Equation 2.1 from Section 2.2, after which the functions start listening, but adds two times the delay to the listening duration, to ensure that the entire packet can be received in the listening window. Finally, the function returns either the received packet, if any, or ***null***.

3.3 Abstract Hardware Simulation

To coordinate the communication between nodes using the hardware interface, we introduce the Coordinator. The Coordinator works by gathering actions (*transmit*, *listen*, *sleep*, and *inform*) sent from nodes using the hardware functions from Section 3.3.2. The pseudo code description of the Coordinator can be seen in Algorithm 8.

3.3.1 Virtual Time

When simulating radio communication it is essential that we factor in time. Time is a large part of a MAC protocol, as is the case with the Slotted ALOHA [10] and LMAC [13] protocols that rely heavily on time slots to be accurate to a point where packets are not dropped because a node starts listening for a synchronisation packet too early or too late. Additionally, as we use an MPI for the emulation of the radio hardware, we need to account for the asynchronous

nature of the MPI. For example, simulating with 50 nodes on a machine with 8 cores would start 51 individual processes, and the operating system would have to schedule accordingly, meaning that not every process (node) would be able to work at all times. Furthermore, as some protocols are designed to conserve power, as the radio hardware might be battery powered, the nodes will spend a large amount of time sleeping, which can create time periods where nothing happens during a simulation. To aid with this, we introduce the notion of virtual time. Virtual time helps us with coordinating the communication between the nodes, as rather than communicating directly with each other using the MPI, nodes communicate through the Coordinator.

For our simulations, virtual time is, essentially, the real-time a node spends during the execution of a MAC protocol, plus the time spent on transmitting, listening or sleeping. As we emulate the radio, transmitting or receiving packets is a matter of sending messages with the MPI, and we do not need to power down the hardware to sleep. To track the virtual time, each node keeps a local real-time clock, measuring the real-time spent during the simulation, and a local time variable. Every time a node either transmits, listens, or sleeps for a duration, the time difference measured by the real-time clock since the node last performed an action, as well as the duration of the transmission, listen, or sleep is added to the local time. This way, any action the node does can be completed almost instantly, and the node may continue executing the simulation, rather than, for example, have to sleep for 10 seconds in real-time, where the process does nothing.

3.3.2 Hardware Functions

In this section, we present the four hardware functions nodes use to communicate with the Coordinator. Common for all four hardware functions is that they all construct an Action object (line 1 in Algorithm 8) and send it to the Coordinator using the MPI. Note that for any action a sent by any node, $a.start \leq a.end$. The hardware functions depend on the following local state. The local state is unique for each node.

$clock \leftarrow$ **now**
$localtime \leftarrow 0$
$id \leftarrow$ unique identifier

In the hardware functions, we utilise a special keyword **now**, which represents the real-time hardware clocks of a node. It is assumed that all clocks run at the same speed. The *clock* variable is used to measure the real-time spent by the node between calls to hardware functions. Initially, we store the current time in the *clock* variable, and use the *clock* to compute the real-time difference between the calling of hardware functions and add the difference to the *localtime* variable (e.g., $localtime \leftarrow (\textbf{now} - clock) + localtime$).
The unique identifier (*id*) of a node is meant to function as the address of a node for passing messages between nodes and the Coordinator. The identifier of the Coordinator = 0, and the identifier of the nodes are in the range $\{1, 2, 3, \ldots, N\}$ for N nodes.
The Transmit (Algorithm 4) function transmits a data packet. The packet is sent to the Coordinator using the MPI, and the Coordinator takes care of distributing the packet to neighbouring nodes listening for packets. The duration of a transmission is computed based on the baud rate (the number of bits the hardware can transmit per second [6]) as well as the size of the packet, using Equation 2.1 found in Section 2.2.
After computing the duration, the transmit action is sent to the Coordinator, the *localtime* variable is set to the end time and the clock is set to **now** before the function returns.

The Listen (Algorithm 5) function takes a duration as input and sends a listen action to the Coordinator. After sending its action, the function waits for a response from the Coordinator

Algorithm 4: The Transmit function.

```
1 Function Transmit(packet)
2     localtime ← (now − clock) + localtime
3     duration ← transmissiontime(|packet|)
4     end ← localtime + duration
5     a ← (transmit, id, localtime, end, packet)
6     send a to Coordinator
7     localtime ← end
8     clock ← now
```

Algorithm 5: The Listen function.

```
Result: A packet or null
1 Function Listen(duration)
2     localtime ← (now − clock) + localtime
3     a ← (listen, id, localtime, localtime + duration, null)
4     send a to Coordinator
5     localtime ← await end from Coordinator
6     packet ← await packet from Coordinator
7     // The packet returned from Coordinator may be null
8     clock ← now
9     return packet
```

at line 5. The **await** keyword is blocking, which means that no other functions can be called from a node while the node is listening for a packet. When the Coordinator processes a listen action, two messages will be sent to the node. The first is the end time, which is assigned to the *localtime* variable, and the second is the packet received (if any). If no packet has been received, the end time received from the Coordinator will be the same as the end time in the action sent to the Coordinator (*localtime* + *duration*), and the packet received will be ***null***. If a packet has been received, the Coordinator will send the time when the packet was received, with the packet following right after.
At most a single packet may be received on a call to the Listen function, but depending on the number of transmissions in the same time interval, no packet could be received, as multiple transmissions either will provide interference for each other, creating collisions, or no transmissions may have happened in the time interval.
After receiving a response from the Coordinator, the function will set the clock variable to **now**, and return either the packet or ***null***.

The Sleep (Algorithm 6) function takes a duration as input and sends a *sleep* action to the Coordinator. As no response is expected of the Coordinator, the function sets the *localtime* and *clock* variables immediately after sending the action.

The Inform function is equivalent to the Sleep function in the sense that it behaves like a call to the Sleep function with the duration set to 0. The function is included in the case none of the other hardware functions is applicable, e.g., in the case where the node is performing longer computations. Regularly informing the Coordinator of a nodes *localtime* will allow the Coordinator to continually process actions from other nodes.

Algorithm 6: The Sleep function.

```
1 Function Sleep(duration)
2     localtime ← (now − clock) + localtime
3     end ← localtime + duration
4     a ← (sleep, id, localtime, end, null)
5     send a to Coordinator
6     localtime ← end
7     clock ← now
```

Algorithm 7: The Inform function.

```
1 Function Inform()
2     localtime ← (now − clock) + localtime
3     a ← (inform, id, localtime, localtime, null)
4     send a to Coordinator
5     clock ← now
```

3.3.3 Coordinator

The pseudo code of the Coordinator is separated into three parts, where each of the parts is repeatedly executed in sequence until the protocol terminates. The first part takes care of receiving actions from nodes, the second part maintains and cleans the *transmits* set of *transmit* actions, and the third part processes, and removes, actions from the *waiting* queue.

The Coordinator works by continuously awaiting actions from any node (line 10). The **await** keyword is blocking, and blocks until any node sends an action to the Coordinator. An action is the 5-tuple Action = (*type*, *source*, *start*, *end*, *packet*). Accessing an element of an action is done with the dot (.) operator, using named access. For example, if node 1 starts listening at time 2 and ends listening at time 5, we have the *listen* action $a = (listen, 1, 2, 5, \textbf{\textit{null}})$. We can access the source of the action using the dot operator as such: $a.source = 1$. The *type* element denotes the type of the action, and is one of either *transmit*, *listen*, *sleep*, or *inform*. The *source* element is the unique identifier of the source node that submitted the action to the Coordinator. The *start* and *end* elements are timestamps for points of time in the execution, where $start \leq end$. Finally, the *packet* element is the data packet sent during transmission or ***null*** for any action where $type \neq transmit$.

Whenever an action is submitted to the Coordinator (line 10), the action is enqueued in the *waiting* queue and added to the *discovered* set. The *waiting* queue is a priority queue of Action objects, where the actions are ordered by their end time, with *transmit* actions before *listen* actions, in case of a tie. The *discovered* set is a set used to track all actions submitted to the Coordinator. The ordering of *sleep* and *inform* in relation to *transmit* or *listen* actions in the *waiting* queue are irrelevant. If the received action is a *transmit* action, the action is also added to the *transmits* set (line 12). Note that any action we receive from a particular node during this part will always have a start time greater than or equal to the end time of the last action received from that node. After receiving an action, the Coordinator continues to the second part.

The *transmits* set is used to gather any *transmit* actions that may cause interference when processing *transmit* actions in the third part of the Coordinator. To make sure that the size of

the *transmits* set does not grow indefinitely, we remove *transmit* actions that will not interfere with future *transmit* actions. To do this we check that the *waiting* queue contains at least one action from each node (line 17). Next, we find the earliest start time of all actions in the *waiting* queue. With this, we can remove any *transmit* actions where the end time is strictly less than the earliest start time found in the *waiting* queue (line 19).

Finally, in the third part of the Coordinator, we process actions from the *waiting* queue. The Coordinator only processes actions when the *waiting* queue fulfils the same condition as in the second part, where the *waiting* queue has to contain at least one action from each node. While this is the case, the Coordinator dequeues a single action (line 24) from the *waiting* queue. Any *sleep* or *inform* actions are processed implicitly, as they only have to be present in the *waiting* queue, to satisfy the condition of the loop. Only the *transmit* and *listen* actions need processing by the Coordinator.

When processing a *transmit* action, the Coordinator first gathers all *transmit* actions from the *transmits* set, that interfere with the *transmit* action being processed (line 27). A *transmit* actions causes interference with another if at any point in time their time intervals intersect (line 28). All applicable *transmit* actions are stored in the *interferers* set, to be used when computing the probability for packet error. Next, the Coordinator iterates through all *listen* actions in the *waiting* queue, and if the time interval of the *transmit* action is fully within the time interval of the *listen* action, the *listen* action is considered relevant for the *transmit* action, and we compute the probability for the packet being received by the listening node (line 33). The probability of the packet being received is computed using the probability for packet error function P_p (Equation 2.10 in Section 2.5). The probability for packet error function is called using the source of the *listen* action l, the source of the *transmit* action a, the *interferers* set, the size of the packet, as well as the end time of the *transmit* action a. Finally, we randomly chose whether the packet should be received by using the computed probability p (line 35). If the listening node has been chosen to receive the packet, the *listen* action l is removed from the *waiting* queue (line 36), and the end time of the *transmit* action a, as well as the packet is sent to the source of the *listen* node.

Processing a *listen* action (line 40) is trivial, as all the Coordinator does is send the end time of the *listen* action and ***null*** to the source of the *listen* action. We do this as dequeuing a *listen* action means that no transmission has been received by the listening node. When the Coordinator responds to the source of the *listen* action, the node is unblocked and may continue executing its protocol.

The complete source code for the C++ implementation can be found on GitHub:
`https://github.com/Joklost/manet-simulations/tree/master/src/coordinator`

Algorithm 8: The Coordinator procedure.

```
1  Action = (type, source, start, end, packet)
2
3  procedure Coordinator()
4      waiting ← priority queue of Action objects, ordered by end time,
5                sleep and inform before transmit, transmit before listen
6      discovered ← empty set of Action objects
7      transmits ← empty set of Action objects
8      repeat
9          // Part 1. Handle incoming message.
10         a ← await Action from any node
11         append a to discovered
12         if a.type = transmit then
13             append a to transmits
14         enqueue a to waiting
15
16         // Part 2. Clean transmits set.
17         if each node has at least one Action in waiting then
18             starttime ← earliest start time of all actions in waiting
19             foreach t ∈ transmits where t.end < starttime do
20                 remove t from transmits
21
22         // Part 3. Process waiting actions.
23         while each node has at least one Action in waiting do
24             a ← dequeue Action from waiting
25             if a.type = transmit then
26                 interferers ← empty set of node identifiers
27                 foreach t ∈ transmits where t ≠ a do
28                     if a.end ≥ t.start and a.start ≤ t.end then
29                         // Transmissions intersect.
30                         append t.source to interferers
31                 foreach l ∈ waiting where l.type = listen do
32                     if a.start ≥ l.start and a.end ≤ l.end then
33                         p ← P_p(l.source, a.source, interferers, |a.packet|, a.end)
34                         shouldreceive ← randomly choose based on p
35                         if shouldreceive then
36                             remove l from waiting
37                             send a.end to l.source
38                             send a.packet to l.source
39             else if a.type = listen then
40                 send a.end to a.source
41                 send null to a.source
42             // sleep/inform actions are handled implicitly.
43     until protocol terminates
```

3.4 Correctness

In this section, we present our correctness arguments for the MPI interface and the Coordinator. We start the section by introducing a set of three invariants for the Coordinator and show why they hold in Section 3.4.1. Next, we go over the methodology for our proof in Section 3.4.2 and an example of a real execution. In Section 3.4.3 we show how a simulated execution of an arbitrary protocol behaves similarly to a real execution. Finally, we summarise the proof in Section 3.4.4.

3.4.1 Invariants

With a set of N unique node identifiers where $nodes = \{1, 2, 3, \ldots, N\}$, the following invariants hold:

1. For all $n \in nodes$, there exists at most one $a \in waiting$ such that $a.source = n$ and $a.type = listen$.
    - There is at most one action with the *listen* type from each node in the *waiting* queue.
2. For all $a, b \in waiting$, if $a.source = b.source$ and $a.type = listen$ then $b.end \leq a.start$.
    - If a node has a *listen* action in the *waiting* queue, no other actions may be present after this.
3. For all $a \in discovered$, if $a.type = transmit$ and there exists $b \in waiting$ such that $b.type = transmit$ and $a.end \geq b.start$ and $a.start \leq b.end$ then $a \in transmits$.
    - Every *transmit* action submitted to the Coordinator that could interfere with any *transmit* action on the *waiting* queue must be present in the *transmits* set.

The first and second invariants are satisfied both by the implementation of the Listen function (Algorithm 5 in Section 3.3.2) and by the Coordinator. The Listen function uses the blocking **await** keyword on line 5 in Algorithm 5, to wait for a response from the Coordinator. The Coordinator, in turn, sends a response to the source of the *listen* action only after the action already has been removed from the *waiting* queue at line 24 or line 36 in Algorithm 8. Due to the blocking nature of the **await** keyword, and the fact that a node is only able to continue execution after a response has been received from the Coordinator, the invariants are satisfied, as it is impossible for a node to send more actions to the Coordinator, while a *listen* action from the same node is already on the *waiting* queue.

The third invariant is satisfied as long as any *transmit* actions that could interfere with any *transmit* action in the *waiting* queue is present in the *transmits* set. With the *discovered* set we can track the *transmit* actions that have been removed from the *waiting* queue, as these could interfere with future *transmit* actions. A *transmit* action a interferes with another *transmit* action b, as long as the time interval for the actions intersects at some point ($a.end \geq b.start$ and $a.start \leq b.end$). At line 19 in Algorithm 8 we have the condition that we only remove *transmit* actions from the *transmits* set if the end time of any action in the set is less than the earliest start time of all actions in *waiting*. Any *transmit* actions fulfilling this condition is safe to remove, as they are no longer be able to intersect with any *transmit* actions in the waiting queue. Additionally, if we have a *transmit* action both in the *waiting* queue and *transmits* set, we will not be able to remove it from the *transmits* set, before the action has been removed from the *waiting* queue, according to the condition at line 19. If we have an action

$t \in \textit{waiting}$ where $t \in \textit{transmits}$ then the earliest start time of all actions in the *waiting* queue is at least $t.\textit{start}$, which means that we will not be able to remove t from the *transmits* set, as $t.\textit{end} \not< t.\textit{start}$, and we know from the implementation of the hardware functions that for any action a, $a.\textit{start} \leq a.\textit{end}$, as shown in Section 3.3.2.

3.4.2 Methodology

Suppose that at a point in time t in a real execution a *listen* actions end, and the listening node receives a packet with some probability p computed, using the packet error probability function P_p. We want to show that in a simulation, with asynchronous virtual-time execution (as implemented in Algorithm 8), the same listening node would receive the packet, with the same computed probability. Figure 13 is a visual representation of a real execution for an arbitrary wireless communication protocol where five nodes communicate through wireless radio broadcasts, without the Coordinator. Vertically, each node has a timeline of actions they perform throughout the execution, where each of the actions is represented as a rectangle (*listen*, *transmit*, or *sleep*) or a line (*inform*). The rectangles and the line represent the start and end time of the actions, where the line has the same start and end time. Each action has an identifier, e.g., t_5^1, and an action is the 5-tuple Action (line 1 in Algorithm 8). The t_5^1 action is a *transmit* action where Node 1 broadcasts some arbitrary data starting at time 2 and ending at time 5: $t_5^1 = (\textit{transmit}, 1, 2, 5, \textit{data})$. The horizontal arrows going from the end of a *transmit* action to a *listen* action represent a packet received by the listening node. If no arrow originates from a *transmit* action, nothing was received of the transmission. In a real execution, a node stops listening after the packet has been received. This is presented in the figure by the l_{14}^4 action, where Node 4 receives a packet from Node 5 at time 12, and stops listening.

The following is the result of processing each of the *transmit* actions from Figure 13, and serve as the timeline of the processing of the *transmit* actions throughout both the real execution, and the simulation:

t_3^4 The packet originating from Node 4 (t_3^4) was dropped by Node 3 (l_6^3) due to interference from Node 1 (t_5^1). Node 2 (l_5^2) started listening too late to receive the packet.

t_5^1 The packet originating from Node 1 (t_5^1) was received by Node 2 (l_5^2) at time 5 but dropped by Node 3 (l_6^3) due to interference from Node 4 (t_3^4).

t_{10}^1 The transmission from Node 1 (t_{10}^1) was not received by any listening nodes, as Node 5 (l_9^5) stopped listening too soon, while Node 3 (l_{10}^3) started listening too late.

t_{10}^2 The packet originating from Node 2 (t_{10}^2) was dropped by Node 3 (l_{10}^3) due to interference from Node 1.

t_{12}^5 The packet originating from Node 5 (t_{12}^5) was received by Node 4 (l_{14}^4) at time 12 but dropped by Node 2 (l_{14}^2) due to distance.

t_{16}^3 The transmission from Node 3 (t_{16}^3) was not received by any listening nodes, as Node 1 (l_{18}^1) started listening too late.

t_{17}^4 The packet originating from Node 4 (t_{17}^4) was dropped by Node 1 (l_{18}^1) due to interference from Node 2 (t_{18}^2), Node 3 (t_{16}^3), and Node 5 (t_{18}^5).

t_{18}^2 The packet originating from Node 2 (t_{18}^2) was dropped by Node 1 (l_{18}^1) due to interference from Node 4 (t_{17}^4), and Node 5 (t_{18}^5).

t^5_{18} The packet originating from Node 5 (t^5_{18}) was dropped by Node 1 (l^1_{18}) due to interference from Node 2 (t^2_{18}), and Node 4 (t^4_{17}).

t^3_{20} The packet originating from Node 3 (t^3_{20}) was received by Node 4 (l^4_{20}).

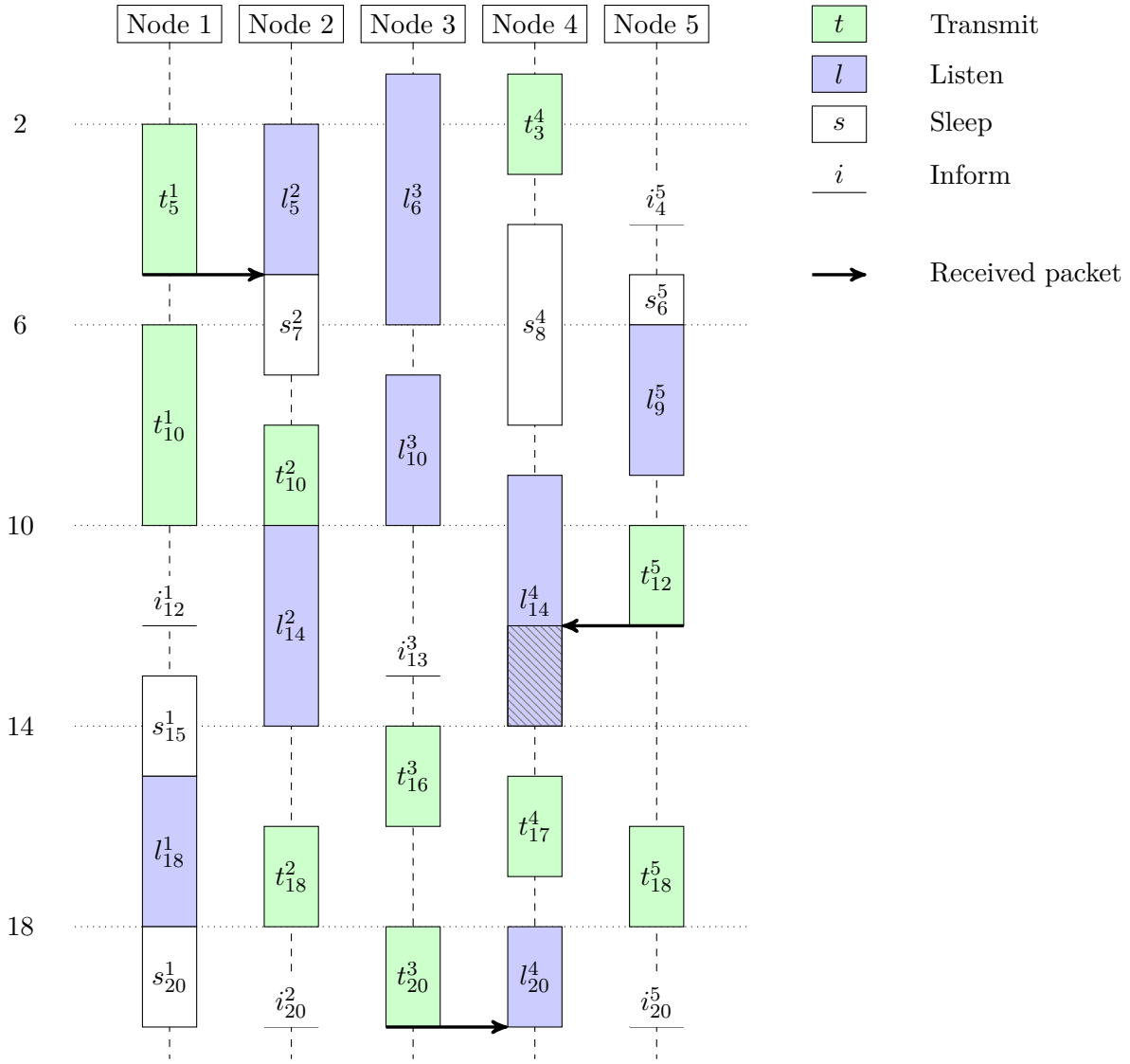


Figure 13: Real execution of an arbitrary protocol with five nodes.

With the invariants outlined above, we want to prove that the packet would be received with the same probability in a virtual-time execution, as it would in a real execution. Due to the asynchronous nature of executing a wireless communication protocol using an MPI, it is not as simple as presenting a single slice of the execution. Instead, to illustrate snapshots of a given point in the execution of the Coordinator we introduce the concept of a cut: A cut consist of two horizontal lines, ⋯⋯ and ——. Everything above the first (⋯⋯) has already been processed by the Coordinator, and everything below the second (——) has not yet been submitted to the Coordinator. We use a cut to show the content of the data structures within

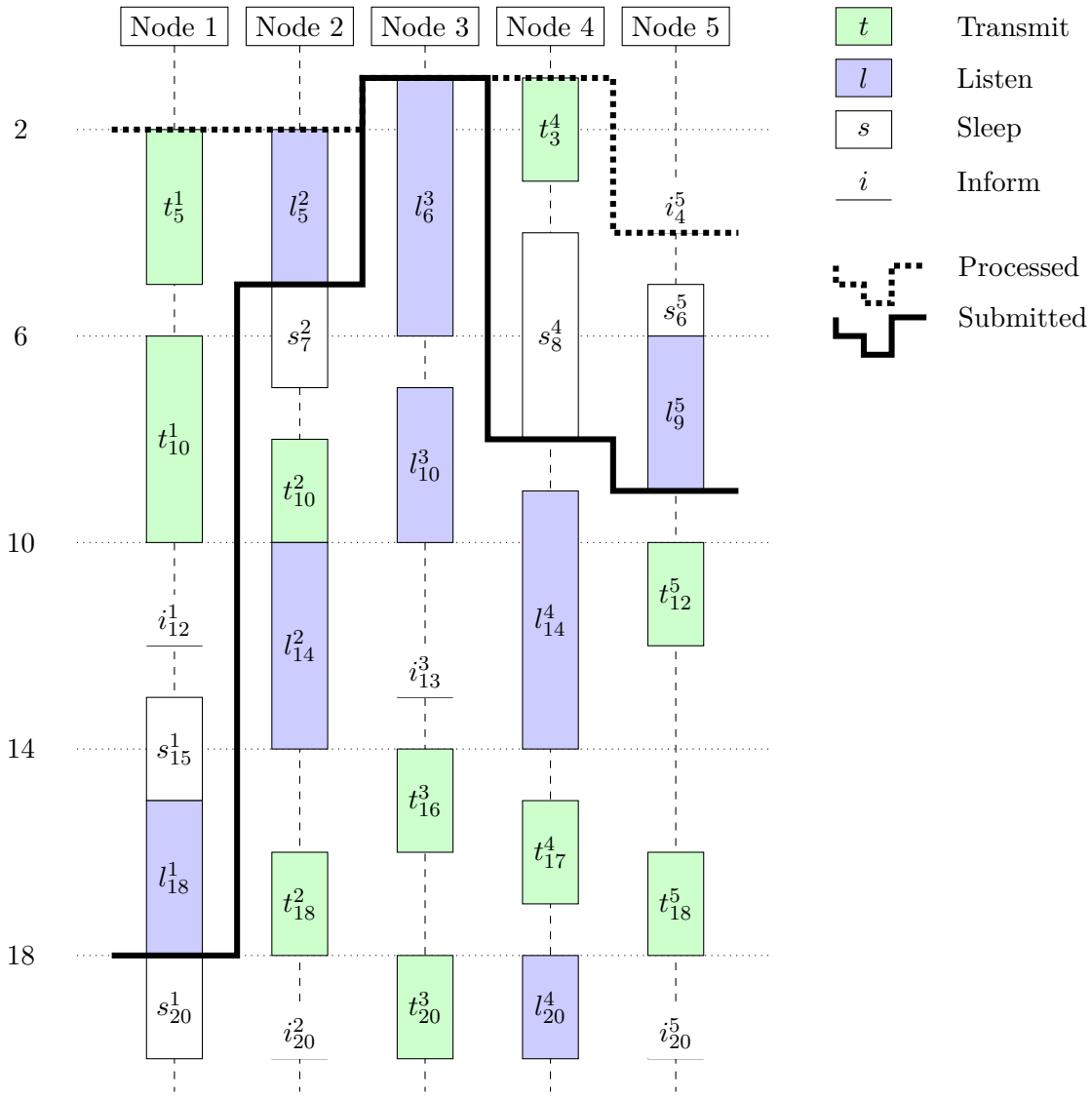
the Coordinator, where we see the order of currently queued actions in the *waiting* queue, as well as the actions in the *transmits* set.

3.4.3 Simulated Execution

Throughout this section we present a simulated, virtual-time, version of the real execution from Figure 13 using the Coordinator from Algorithm 8. In a simulated execution, the nodes communicate through the Coordinator, rather than directly with each other. Throughout the figures in this section, it is assumed that all nodes interact with the Coordinator implicitly. We utilise cuts to present different possible scenarios of the Coordinator: No action can be processed, processing a *transmit*, processing an *inform* action, processing a *listen* action, and finally, processing a *sleep* action. For each of the different scenarios, we start with a concrete example, basing it on the accompanying figure, and ending with a generalisation of the scenario. When processing each of the *transmit* actions we follow the descriptions listed in Section 3.4.2.

Cut 1: Nothing can be processed

For the first cut, we have a scenario where nothing may be processed. Recall that the condition for Part 3 of the Coordinator procedure is that the waiting queue is only processed if all nodes have at least one action on the queue. As of this cut, Node 3 has yet to submit an action to the Coordinator which means that the Coordinator is unable to progress from this point until the l_6^3 action is submitted at some point, later in the execution. Note that due to the asynchronous nature of our simulation, and the fact that only *listen* actions are blocking on the hardware side, it is very possible to have a scenario, where a node has submitted a large number of actions, as Node 1 has in the figure, where Node 1 is currently waiting for the l_{18}^1 action to be processed, and Node 3 has yet to submit any actions to the Coordinator. Additionally, nodes may be doing other work internally, before submitting more actions to the Coordinator, which is why Node 4 might not have submitted the *listen* action l_{14}^4 as of this cut. Node 1, Node 2, and Node 5 are not able to submit more actions to the Coordinator, before the *listen* actions they have submitted have been processed.



The content of the *waiting* queue and *transmits* set as of this cut is:

$waiting \leftarrow \left\langle t^4_3, i^5_4, t^1_5, l^2_5, s^5_6, s^4_8, l^5_9, t^1_{10}, i^1_{12}, s^1_{15}, l^1_{18} \right\rangle$

$transmits \leftarrow \left\{ t^4_3, t^1_5, t^1_{10} \right\}$

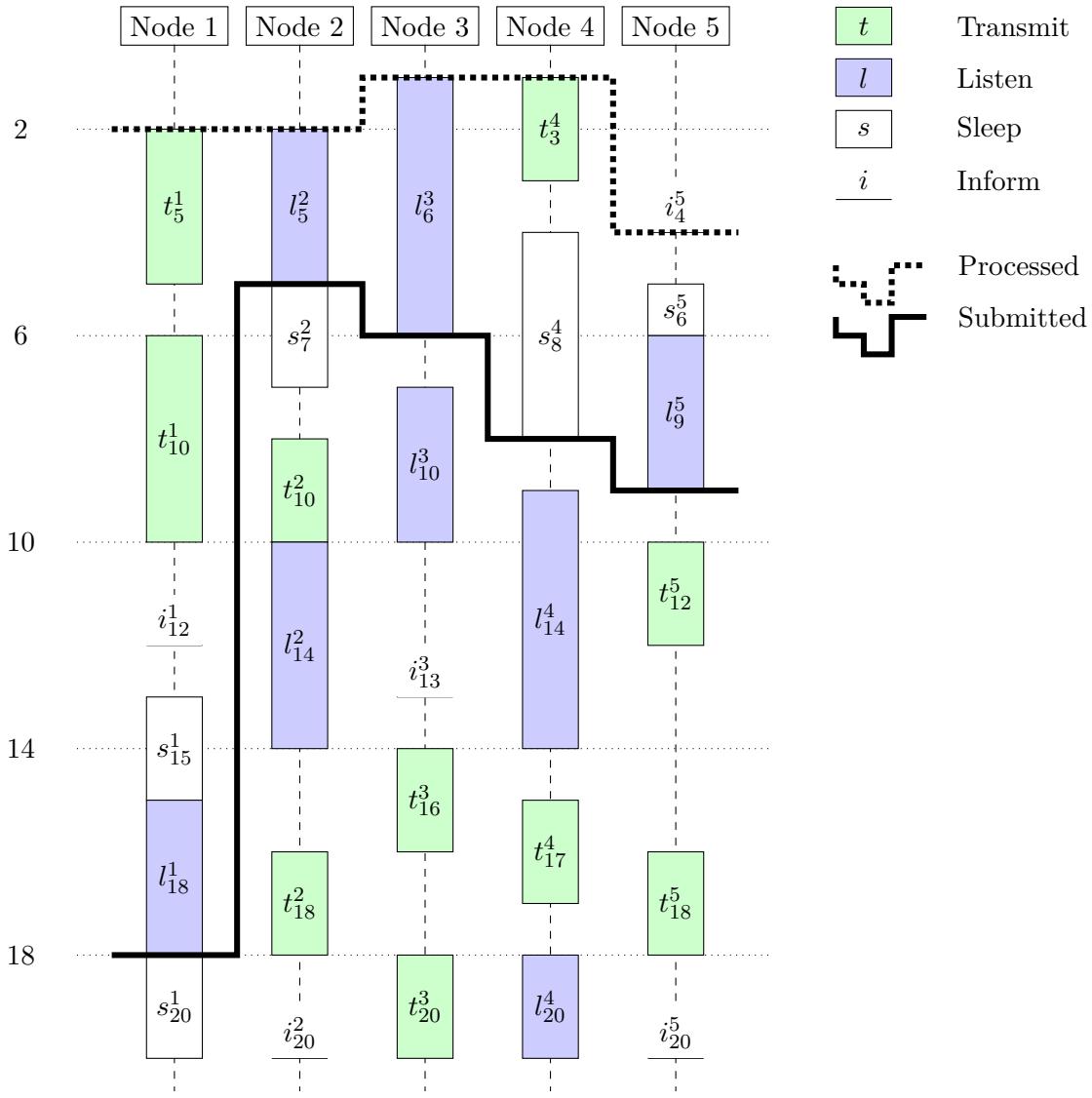
The *waiting* queue is a priority queue where actions are ordered by end time, and *sleep* and *inform* action must be before *transmit* actions, which must be before *listen* actions, if the end time is the same.

Figure 14: Waiting for Node 3 to submit an action.

Cut 2: *transmit* action

The next cut is a snapshot of the Coordinator directly after Node 3 has submitted the l^3_6 action. With this action in the *waiting* queue, the Coordinator may begin to process the actions in the *waiting* queue and *transmits* set. First, the Coordinator would check if any *transmit* actions should be removed from the *transmits* set, but as none of the actions in the set has a start time earlier than the start time of the earliest action in the *waiting* queue (l^3_6 or t^4_3, both with start =

1) nothing can be removed. Next, the Coordinator can begin processing actions in the *waiting* queue.



The content of the *waiting* queue and *transmits* set as of this cut is:

$$\textit{waiting} \leftarrow \left\langle t^4_3, i^5_4, t^1_5, l^2_5, s^5_6, l^3_6, s^4_8, l^5_9, t^1_{10}, i^1_{12}, s^1_{15}, l^1_{18} \right\rangle$$

$$\textit{transmits} \leftarrow \left\{ t^4_3, t^1_6, t^1_{10} \right\}$$

Figure 15: Processing a *transmit* action.

The first action in the *waiting* queue is the *transmit* action t^4_3. Recall that when processing a *transmit* action, the Coordinator iterates through the *transmits* set to find other *transmit* actions with intersecting time intervals, and that invariant 3 ensures that any *transmit* action with an intersecting time interval will be in the *transmits* set. In this case, only the t^1_5 action intersects with the t^4_3 action, so the source of that action is included in the *interferers* set. Next, the Coordinator iterates through all *listen* actions in the *waiting* queue, and only if the time interval for the *transmit* action is fully within the time interval of any *listen* action, we

compute the probability for packet error on a transmission between the source of the *listen* action ($l_6^3.source$) and the source of the *transmit* action ($t_3^4.source$). With the probability for packet error, Equation 2.10 in Section 2.5, $p = P_p(l_6^3.source, t_3^4.source, \{t_5^1.source\}, |t_3^4.packet|, t_3^4.end)$, we can randomly choose whether the transmission should be received, or dropped, and either finish processing the *listen* action by sending the packet to the source of the action, or move on to the next *listen* action. For the t_3^4 action, we assume the packet to be dropped by the listening node.

When processing the t_3^4 action we have two cases to consider for this particular scenario: In the first case, the packet is received by Node 3 and the l_6^3 action is removed from the *waiting* queue. Should this be the case, the condition for processing actions in the *waiting* queue is no longer satisfied, and the Coordinator will not be able to process any further actions until the l_{10}^3 action has been submitted to the Coordinator. In the second case, the packet is dropped and not received by Node 3. In this case, the l_6^3 action is still on the *waiting* queue, and the Coordinator can continue processing actions as the *waiting* queue still contains at least one action from each node.

This holds in general, as every *listen* action will be after any *transmit* action t on the *waiting* queue, and every *transmit* action that could interfere with t is in the *transmits* set, as stated by invariant 3. Let us assume that a node n submitted the *transmit* action t, and that t is at the head of the *waiting* queue. When processing t, we want to compute the probability for any node m listening in the time interval of the *transmit* action t to receive the packet transmitted by n. When this is the case, the *listen* action is relevant to t. To do this, we need to make sure that any *listen* action, that is relevant for the processing of t is on the *waiting* queue, and any *transmit* actions that could interfere with t is in the *transmits* set, when processing t.

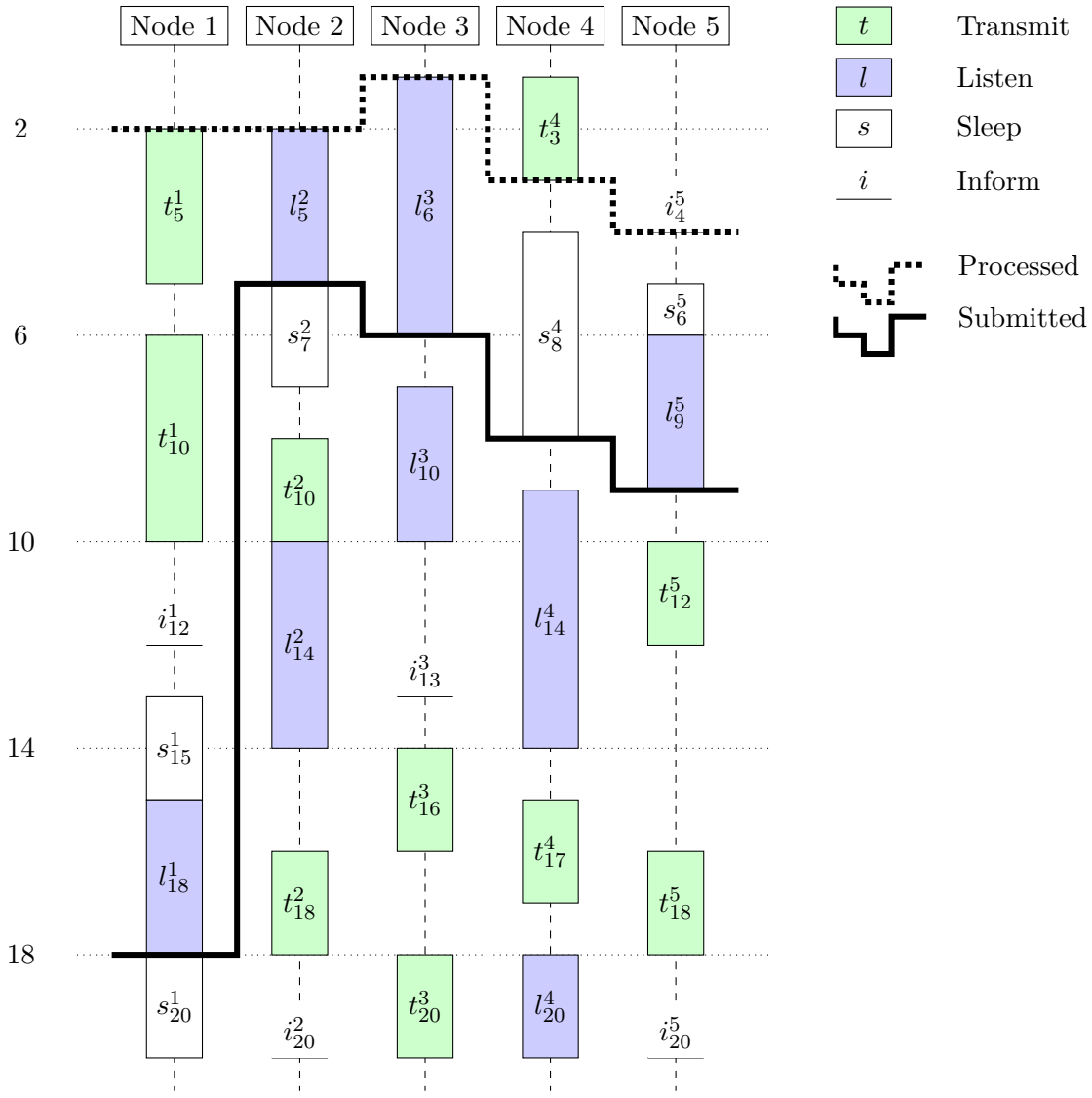
First, we know that every *listen* action l that could receive the *transmit* action t is on the *waiting* queue, as the end time of any l on the *waiting* queue, when processing t, has to be greater than, or equal to, the end time of t. Recall that due to the ordering of the *waiting* queue, we always process any *transmit* action before processing a *listen* action. In addition to this, when processing t, we know that for all *listen* actions l, where the time interval of t is fully within the time interval for l, the source node m of l will have only the l action on the *waiting* queue. We know this as any action a from node m would have an end time earlier than the start time of l, $a.end \leq l.start$, as stated by invariant 1, which means that the a action would already have been processed before we process t. Additionally, invariant 2 states that if a node has a *listen* action in the *waiting* queue, no other actions may be present after this. Finally, as we know that the condition for processing actions in the *waiting* queue is satisfied when we begin processing the *transmit* action t, any node m that does not have a *listen* action that is relevant to t, is unable to submit a *listen* action that could become relevant to t, as any future action submitted from m would have a start time greater than, or equal to, the end time of the previous action submitted from m, which we know is greater than the end time of the action t that is currently being processed.

With these two points, when processing a *transmit* action t, we can safely compute the probability for any *listen* action l relevant to t, including the interference from any interfering *transmit* actions in the *transmits* set.

Cut 3: *inform* action

The next cut is a snapshot of the Coordinator directly after the t_3^4 action has been processed and removed from the *waiting* queue. The condition for processing actions remains satisfied, as there is still at least one action from each node in the *waiting* queue. At the head of the *waiting*

queue is the *inform* action i_4^5. Processing an *inform* action is trivial, as no processing is needed for this action. The action is simply removed from the *waiting* queue, and the Coordinator may move on to the next action if the condition is still satisfied. This holds in general for any *inform* action.



The content of the *waiting* queue and *transmits* set as of this cut is:

$$\textit{waiting} \leftarrow \left\langle i_4^5, t_5^1, l_5^2, s_6^5, l_6^3, s_8^4, l_9^5, t_{10}^1, i_{12}^1, s_{15}^1, l_{18}^1 \right\rangle$$

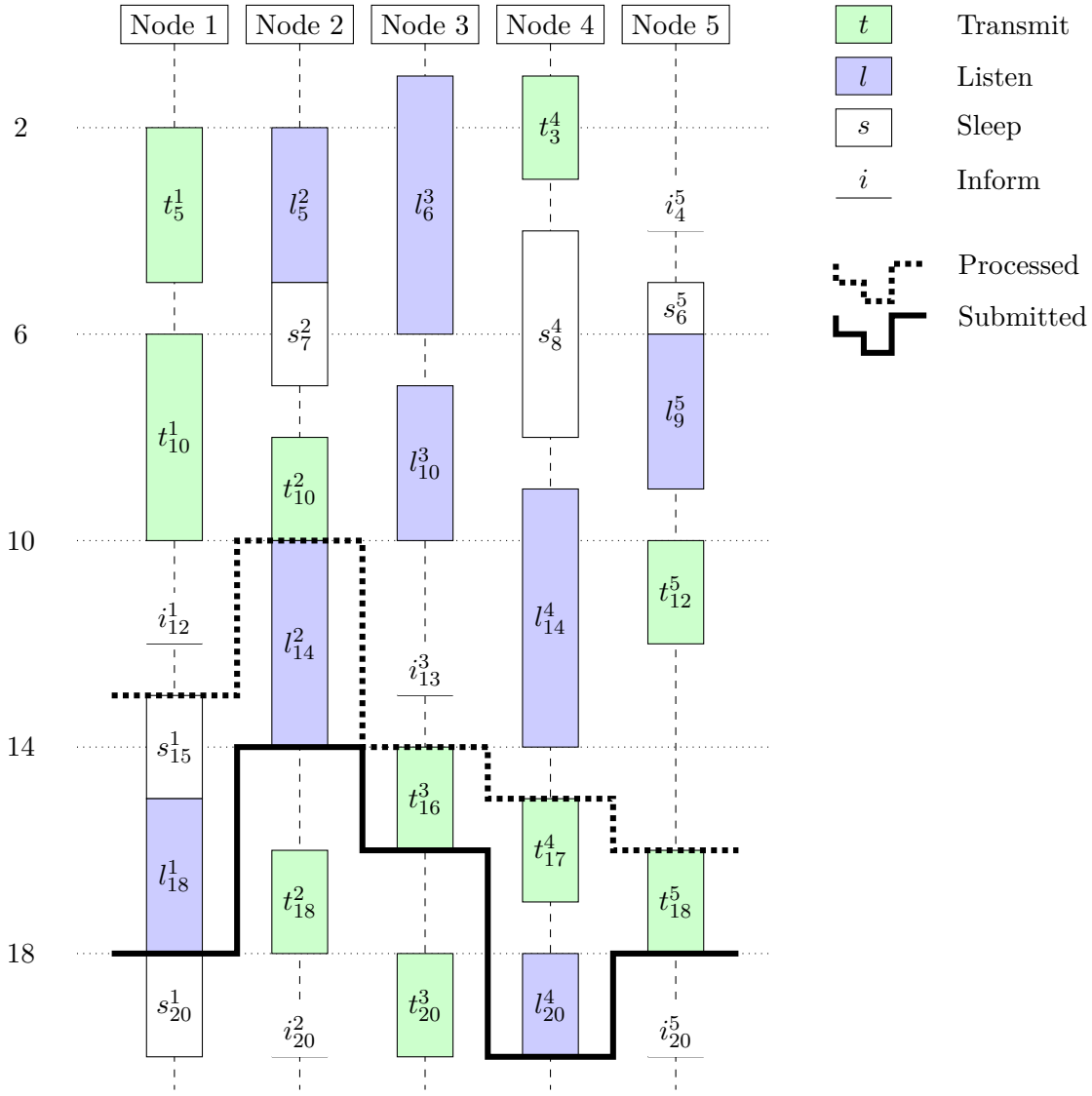
$$\textit{transmits} \leftarrow \left\{ t_3^4, t_6^1, t_{10}^1 \right\}$$

Figure 16: Processing an *inform* action.

Cut 4: *listen* action

This cut shows a snapshot of the Coordinator directly after the t_{16}^3 action has been submitted. With this action on the *waiting* queue the condition for processing actions has been satisfied, and the Coordinator may process the first action on the queue, the l_{14}^2 action. For this cut,

there are two interesting points to note. First, when a *listen* action is at the head of the *waiting* queue, it means that no packet has been received during this transmission. When this is the case, the action is removed, and ***null*** is sent to the source of the *listen* action. Second, the l_{14}^4 action has already been processed, and removed from the *waiting* queue, even though the action could not have been processed before the t_{16}^3 action had been submitted. The l_{14}^4 action was removed from the *waiting* queue early, as the node had received a packet when processing the t_{12}^5 action.



The content of the *waiting* queue and *transmits* set as of this cut is:

$$waiting \leftarrow \left\langle l_{14}^2, s_{15}^1, t_{16}^3, t_{17}^4, t_{18}^5, l_{18}^1, l_{20}^4 \right\rangle$$

$$transmits \leftarrow \left\{ t_{10}^1, t_{10}^2, t_{12}^5, t_{16}^3, t_{17}^4, t_{18}^5 \right\}$$

Figure 17: Processing a *listen* action.

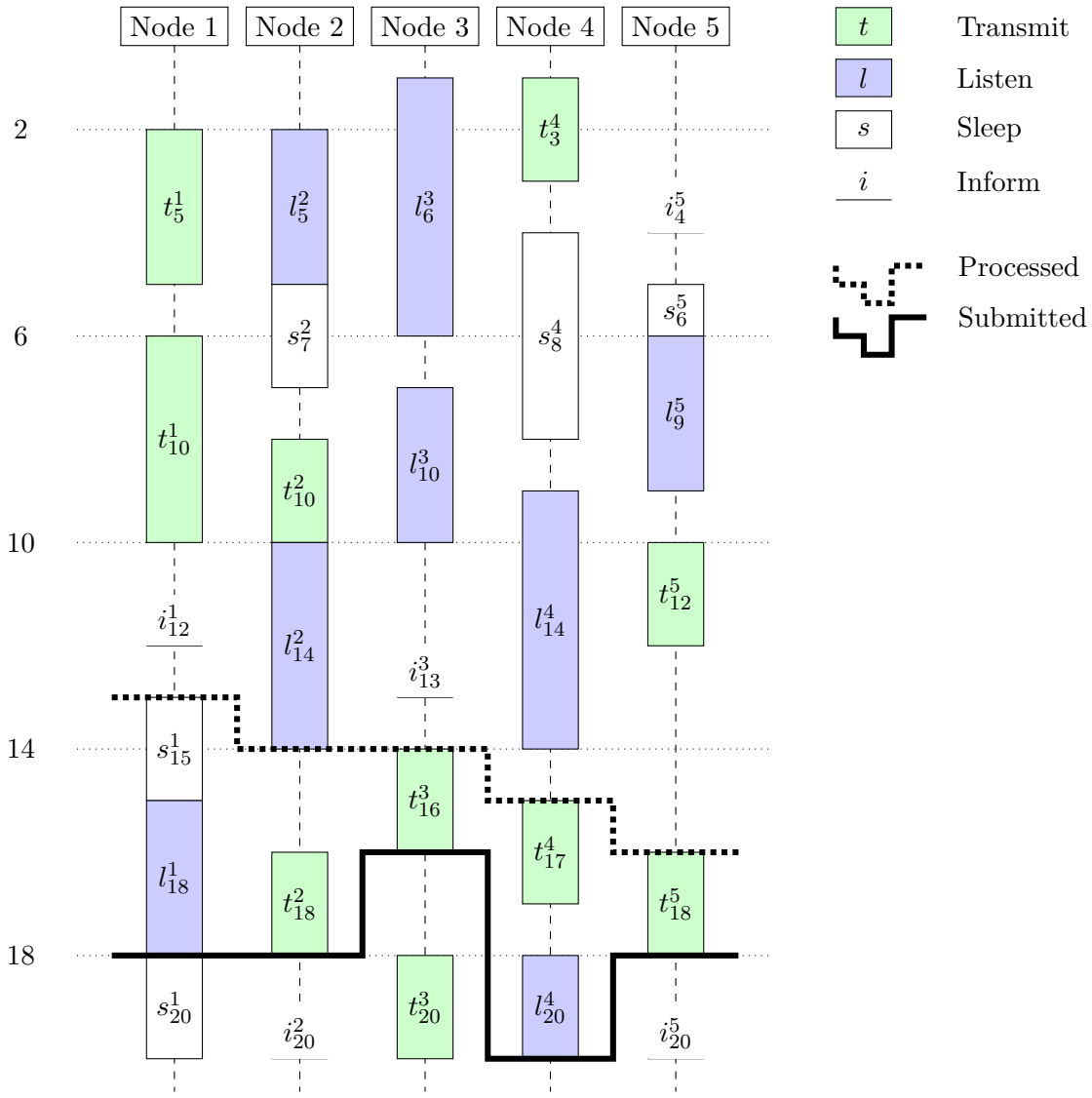
This holds in general, as no *transmit* action can be fully within the time interval of any *listen* action, if the *listen* action is at the head of the *waiting* queue. The *listen* action would have been

removed when processing earlier *transmit* actions, as *transmit* actions are ordered before *listen* actions in the *waiting* queue. Let us assume that a node n submitted the *listen* action l, and that l is at the head of the *waiting* queue. When processing the l action, no actions from node n before, or after, l is present in the *waiting* queue, as all previous actions have been processed. Additionally, no *transmit* actions from any other node m can be present on the *waiting* queue, with end time less than, or equal to, the end time of the l action, as these would already have been processed, due to the ordering of the priority queue.

Cut 5: *sleep* action

The final cut is a snapshot of the Coordinator directly after the t_{18}^2 action has been submitted to the Coordinator. The t_{18}^2 action was submitted by Node 2 after the l_{14}^2 action had been processed without the node receiving any data, and the *transmit* action enabled the Coordinator to continue processing actions from the *waiting* queue. The head of the *waiting* queue is the *sleep* action s_{15}^1, which, similarly to an *inform* action, is trivial to process, as no processing is needed for the Coordinator. Again, similarly to an *inform* action, this holds in general for any *sleep* action.

After removing the l_{14}^2 action from the *waiting* queue, the Coordinator was able to remove the t_{10}^1, t_{10}^2, and t_{12}^5 actions from the *transmits* set, as the new earliest start time is now the start time of the s_{15}^1 action.



The content of the *waiting* queue and *transmits* set as of this cut is:

$$waiting \leftarrow \left\langle s^1_{15}, t^3_{16}, t^4_{17}, t^5_{18}, t^2_{18}, l^1_{18}, l^4_{20} \right\rangle$$

$$transmits \leftarrow \left\{ t^3_{16}, t^4_{17}, t^2_{18}, t^5_{18} \right\}$$

Figure 18: Processing a *sleep* action.

3.4.4 Summary

Throughout this section, we have presented arguments of correctness for the Coordinator and the four hardware functions, by analysing each of the five cases for the Coordinator processing an action, starting with a concrete example, based on the accompanying figure, and finishing each of the scenarios with a generalisation.

Chapter 4

Experiments

In this chapter, we present the LMAC protocol [13], as well as some fragments of a C++ implementation of the protocol, in Section 4.1. In Section 4.2 we present a scalability experiment of the MPI and the Coordinator, running with the LMAC protocol.

4.1 LMAC

LMAC [13][2] is a Time Division Multiple Access (TDMA) protocol designed to be lightweight and energy efficient, to prolong the lifetime of a network. In the protocol, time is divided into frames, that consist of a fixed number of time slots. Each node in a network controls a single time slot in each frame, and if a node has any data to transmit, the node waits for its time slot to come up, which means that a node can transmit the data without causing a collision, or interference for other nodes. Additionally, whenever a nodes time slot comes up, the node transmits a short synchronisation packet at the beginning of the time slot. For every other time slot, the node listens for the synchronisation packet from other nodes, to maintain synchronisation and keep neighbourhood information up-to-date. The structure of a synchronisation can be found in Figure 19.

The LMAC protocol consists of four phases:

Initialisation
: Each node initially starts in the Initialisation phase. In this phase, the node has yet to choose a time slot, so instead, it listens for synchronisation packets in every time slot. When a synchronisation packet has been received and a neighbouring node has been detected, the node synchronises, and the node knows the current slot number. After a synchronisation packet has been received, the node switches to the Wait phase at the beginning of the next frame. A single node chosen as the gateway node starts the Initialisation phase by picking a time slot and proceeding to the active phase.

Wait
: A node in the Wait phase waits for a random amount of frames (up to a pre-defined limit) between receiving the synchronisation packet and moving to the Discover phase.

Discover
: In the Discover phase, the node collects first order neighbourhood information from neighbouring active nodes, by listening for synchronisation packets throughout one frame, and recording the occupied time slots. Once a frame worth of neighbourhood information has

been recorded, the node chooses a random, available, time slot, and proceeds to the Active phase.

Active

Finally, a node in the Active phase can transmit a data packet in its chosen time slot, while listening in other time slots to accept data from neighbouring nodes. The node still uses the synchronisation packet to keep neighbourhood information up-to-date, and attempts to detect, and report, possible collisions in the network. When a node in the Active phase is informed of a collision in its chosen time slot, the node will give up its time slot, and proceed to the Wait phase. A collision happens when two or more nodes have chosen the same time slot. Nodes that are part of a collision is unable to detect the collision, and they need to be informed by their neighbours. When a node has detected a collision, the time slot in question is included in the synchronisation packet that will be sent from the node in its next time slot, to inform all neighbours of the collision.

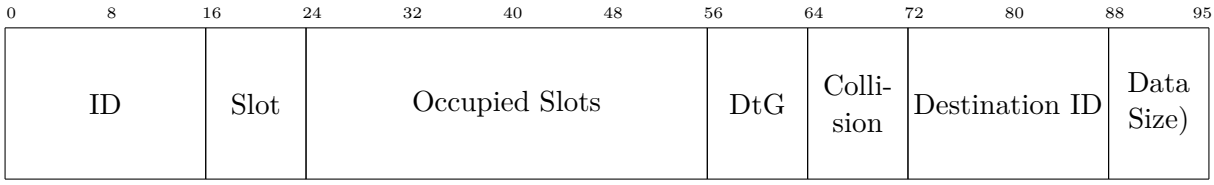


Figure 19: The synchronisation packet structure in LMAC [13, p. 2].

In LMAC, each node keeps track of its *hop-distance* to the pre-defined gateway node [13] and includes this hop-distance in the synchronisation packet as the "DtG", or distance to gateway, field. When an Active node has a data packet to transmit, the node looks through its neighbourhood information to find a neighbouring node that is closest to the gateway, pick this node as the destination for its message, and include the destination in the synchronisation packet. Should multiple neighbours nodes be equally close to the gateway, a destination will randomly be picked between them. If a destination, and a data size, is included in the synchronisation packet, and the destination is equal the id of the node, the node will listen for a data packet, in the time slot, after having received the synchronisation packet. Additionally, it is only possible for a node to transmit a single data message per frame and the maximum size of a data packet is 256 bytes.

(a) Grid topology synchronisation.

(b) Grid topology routing.

(c) Dynamic topology synchronisation.

Figure 20: LMAC topology synchronisation and routing.

Figure 20 contains YouTube links to three visualisations of the LMAC protocol, where Figure 20a visualises node synchronisation and network stabilisation, Figure 20b visualises the routing from a data generation node to the gateway node, and Figure 20c visualises part of the node synchronisation of a larger network. Nodes are coloured per their phase, with white nodes being in the Initialisation phase, red nodes in the Wait phase, blue nodes in the Discover phase, and green nodes in the Active phase. When the node enters the Active phase, the chosen slot is drawn on the node.

In the routing visualisation in Figure 20b, the bottom left node is chosen as the gateway node, and the top right node generates a single data packet each frame. An arrow originates from a node whenever a message is sent from the node, with a green arrow denoting the synchronisation packet, and the red arrow denoting a data packet.

Code 1 shows a snippet of the LMAC implementation for the part of the code where a node in the Active phase constructs and broadcast the synchronisation packet, and a data packet, if any has been received, in its chosen time slot. If the node has a data packet to send, it computes the receiver with the lowest hop-distance to the gateway, and adds it to the synchronisation packet, along with the size of the data packet.

```
...
/* Create synchronisation signal. */
ControlPacket ctrl{id, state.chosen_slot,
                   state.occupied_slots, state.gateway_distance,
                   state.collision_slot, receiver, data_size};
state.collision_slot = NO_SLOT;
/* Send initial synchronisation signal. */
hardware::sleep(3ms);
hardware::broadcast(mpilib::serialise(ctrl));
/* Send packet, if any. */
if (!state.data_packet.empty()) {
    hardware::sleep(10ms);
    hardware::broadcast(state.data_packet);
    state.data_packet.clear();
}
...
```

Code 1: Construct and send synchronisation and data packets.

Code 2 shows another snippet of the LMAC protocol implementation, where a node in either the Initialisation, Discover, or Active phase is listening for synchronisation packets. When a node receives the synchronisation packet, the node updates its local state, and the neighbourhood information, with any new information, received in the synchronisation packet. If a packet as destined for the node, and the node is in the Active phase, the node listens for the data packet, and stores it for later transmission.

The complete source code for the C++ implementation can be found on GitHub:
https://github.com/Joklost/manet-simulations/tree/master/src/lmac

```
...
/* Listen for synchronisation signal. */
auto ctrl_data = hardware::receive(20ms);
if (!ctrl_data.empty()) {
    auto ctrl = mpilib::deserialise<ControlPacket>(ctrl_data);
    state.occupied_slots[ctrl.chosen_slot] = true;
    if (ctrl.gateway_distance + 1 < state.gateway_distance) {
        state.gateway_distance = ctrl.gateway_distance + 1;
    }
    /* Update neighbourhood information. */
    ...
    if (state.phase == active) {
        if (ctrl.destination_id == id && ctrl.data_size > 0) {
            /* Listen for packet. */
            auto data = hardware::receive(70ms);
            if (!data.empty()) {
                state.data_packet = data;
                ...
            }
        }
    }
}
...
```

Code 2: Receive synchronisation and data packets.

4.2 Scalability

We conducted two scalability experiments with the LMAC protocol from Section 4.1. The first experiment is to show how simulations with the centralised Coordinator performs, with a fixed number of nodes, when scaling the amount of CPUs available for the simulation. In the second experiment, we attempt to scale the number of nodes, while also scaling the number of CPUs to match the size of the network.

Cores	16	32	64
Time	41:49 min	34:52 min	19:28 min

Table 1: 5-minute random walk topology, 60 nodes.

In the first experiment, we simulate the LMAC protocol with 60 nodes in a generated random walk topology, over a 5-minute real-time period with approximately 300 links at all times, throughout the log. The results for this experiment can be seen in Table 1. The experiment demonstrate that scaling the amount of CPUs will significantly improve the time required to run the simulation, but it also show that the simulation does not scale very well with a larger number of nodes, as even when running the simulation with 64 cores available, simulating the experiment takes way longer than the 5-minute GPS log we simulate.

Nodes	100	300	600
Links	≈450	≈1200	≈2400
Cores	128	320	576
Time	45:43 min	6:51:53 hrs	...

Table 2: 5-minute random walk topology.

The second experiment further shows the scalability issues of the Coordinator. In this experiment, we attempted to simulate the LMAC protocol while scaling the number of nodes in the network, as well as the amount of CPUs available. The results for this experiment can be seen in Table 2. A 5-minute real-time simulation with 100 nodes, and approximately 450 links, took over 45 minutes with 128 cores, and a 300 node experiment, with approximately 1200 links, took almost seven hours using 320 core. After this result, we decided not to continue the experiment.

The annotated random-walk GPS logs can be found on GitHub:
`https://github.com/Joklost/manet-simulations/tree/master/src/coordinator/logs`

Chapter 5

Conclusion

In this thesis, we introduce a C++ library for writing and running, simulations of the MAC protocols behind the mesh communication in a MANET using an MPI. Our library consists of X parts: A hardware interface header file, used for writing implementations of MAC protocols, that emulate the physical part of a device in a MANET, a Coordinator, facilitating and coordinating the communication between the emulated devices, and finally the link model, where we can annotate network topology GPS logs with links between nodes, based on a model for link path loss, where we approximate the path loss for a link using building footprints between the nodes of a link, on OpenStreetMap map tiles. With the annotated GPS logs, we can simulate wireless radio communication, through the Coordinator, where we can simulate packet errors and collisions caused by interfering transmissions or bad links.

In Section 2.3 and Section 2.4, we propose a link modelling method for approximating the path loss on a link by computing the percentage amount of building between the two nodes of a link, and show that the computed RSSI values are roughly equivalent to field measurements, that is more reliable than computing the path loss entirely based on the distance.

Section 3.4 presents our arguments of correctness for the Coordinator and the hardware functions, analysing the different possible cases for when the Coordinator processes an action, starting with a concrete example, and finishing with a generalisation of each of the scenarios.

In Chapter 4, we present the LMAC protocol and show how our C++ library can be used to simulate the protocol, and we present the results of our scalability experiments for the Coordinator. We do, however, face significant scalability problems, as we rely on a single centralised Coordinator, as shown in Section 4.2. Our experiments show that we can simulate 100 nodes in about 45 minutes, while using 128 cores, and that simulation time scales significantly with an increasing number of nodes.

5.1 Future Work

Running simulations with a large number of nodes pose a significant scalability challenge, as shown in Section 4.2. The reason behind this is very likely the fact that we use a centralised Coordinator, to coordinate the communication between a large number of nodes, using an MPI.

When visualising with the Visualiser tool, we would ideally like to visualise a greater number of nodes than what is currently supported, as visualising protocol and communication logs can be a significant help when debugging MAC protocol implementations.

Finally, we would like to be able to create a better path loss model, by using more data to approximate the link model. The current version of the model is not very accurate when it comes to the path loss for buildings, as the logs we had access to did not have enough viable data.

Chapter 6

Bibliographical Remarks

As this project, and this thesis, was part of a multi-semester project, we have re-used parts of our thesis from last semester [3]. The following is a list of the sections that include parts of the last thesis, and the changes that has been made to them in this thesis:

- Parts of the summary on page v is based on text from other sections in this thesis.
- The abstract on page iv has been adapted from the abstract on page iii in [3] to reflect the contributions and results of this project.
- Chapter 1 (excluding the figure) has been adapted from Chapter 1 in [3] to reflect the contributions of this project.
- Section 1.1 has been included from [3].
- The introduction of Chapter 2 is an adapted version of the introduction to Chapter 2 in [3].
- Section 2.3 has been adapted, and shortened, from Section 2.1.2 in [3].
- Section 2.5 has been adapted with slight changes to the naming in some of the equations, to the presented example, as well as to include time for the RSSI function, to Section 2.2 in [3]. Additionally, the graphs showing the probability for packet error has been updated slightly.
- Section 3.3.1 includes parts of Section 3.1.1 from [3].
- Section 3.3.2 has been adapted from Section 3.1.3 in [3], to conform to new behaviour. Additionally, two of the functions have been renamed.
- Section 3.3.3 is a heavily modified version of Section 3.2 in [3]. The name and functionality of the Coordinator have changed greatly from the Centralised Controller of [3], but the Coordinator still has basis in this section.
- The beginning of Chapter 5 has been adapted from Chapter 5 in [3].

Bibliography

[1] Rasmus Liborius Bruun. *Mobile ad-hoc network performance assessment based on simulation with scenario specific propagation and multi-link modelling.* Aalborg University, 2018.

[2] Ansgar Fehnker, Lodewijk van Hoesel, and Angelika Mader. Modelling and verification of the lmac protocol for wireless sensor networks. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods*, pages 253–272, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[3] Charlie Dittfeld Byrdam Jonas Kloster Jacobsen. *Message Passing Interface for Simulation of Mobile Ad-hoc Networks.* Aalborg University, 2019.

[4] mapbox. Our map data. https://docs.mapbox.com/help/how-mapbox-works/mapbox-data/. Accessed: 6. June 2019.

[5] P. Massoud Salehi and J. Proakis. *Digital Communications.* McGraw-Hill Education, 2007.

[6] MathWorks. Baudrate. https://mathworks.com/help/matlab/matlab_external/baudrate.html. Accessed: 8. January 2019.

[7] A. Merts and A. Barnard. Simulating manets: A study using satellites with aodv and anthocnet. In *2016 Pattern Recognition Association of South Africa and Robotics and Mechatronics International Conference (PRASA-RobMech)*, pages 1–5, Nov 2016.

[8] Metageek. Rssi vs dbm. https://www.metageek.com/training/resources/understanding-rssi.html. Accessed: 8. January 2019.

[9] Morteza Mohammadi Zanjireh and Hadi Larijani. A survey on centralised and distributed clustering routing algorithms for wsns. volume 2015, 05 2015.

[10] Lawrence G. Roberts. Aloha packet system with and without slots and capture. *SIGCOMM Comput. Commun. Rev.*, 5(2):28–42, April 1975.

[11] Marjan Sirjani, Ali Movaghar-Rahimabadi, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using rebeca. *Fundam. Inform.*, 63:385–410, 2004.

[12] Aalborg University. mccaau. https://sites.google.com/site/mccaau/home, 2018. Accessed: 10. January 2019.

[13] L.F.W. van Hoesel and Paul J.M. Havinga. A lightweight medium access protocol (lmac) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches. In *1st International Workshop on Networked Sensing Systems (INSS)*, pages 205–208. Society of Instrument and Control Engineers (SICE), 2004.

[14] Behnaz Yousefi, Fatemeh Ghassemi, and Ramtin Khosravi. Modeling and efficient verification of broadcasting actors. In *FSEN*, volume 9392 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2015.

[15] Behnaz Yousefi, Fatemeh Ghassemi, and Ramtin Khosravi. Modeling and efficient verification of wireless ad hoc networks. *Formal Asp. Comput.*, 29(6):1051–1086, 2017.

[16] Fengling Zhang, Lei Bu, Linzhang Wang, Jianhua Zhao, Xin Chen, Tian ZHANG, and Xuandong Li. Modeling and evaluation of wireless sensor network protocols by stochastic timed automata. *Electronic Notes in Theoretical Computer Science*, 296:261–277, 08 2013.

Glossary

baud rate

The baud rate is the rate at which information can be transferred as a wireless signal [6], and is equivalent to bits per second. 11, 12, 25

path loss

The signal loss inflicted by the propagation of a electromagnetic wave from transmitter to receiver [1, p. 10]. iii, v, 3, 5, 7, 11, 12, 13, 14, 15, 47, 48

Acronyms

BOPL
Building Obstructed Path Loss. 14, 16, 17

CVPL
Clear View Path Loss. 14, 16

dB
decibel. 19, 55

dBm
dB relative to a milliwatt. 11, 19, 20, 21

GPS
Global Positioning System. 8, 9, 10, 12, 44, 45, 47

LMAC
Lightweight Medium Access Protocol. v, 3, 10, 41, 42, 43, 44, 45, 47

MAC
Medium Access Control. iii, v, 24, 25, 47, 48

MANET
Mobile Ad-hoc Network. iii, v, 3, 5, 7, 8, 12, 47

MPI
Message Passing Interface. iii, v, 3, 23, 24, 25, 30, 32, 41, 47, 48

RSSI
Received Signal Strength Indication. 7, 9, 10, 11, 12, 13, 14, 15, 17, 19, 20, 21, 47, 49

SN(I)R
Signal to Noise (and Interference) Ratio. 19, 20

SNR
Signal to Noise Ratio. 19

TDMA

Time Division Multiple Access. 41

WSN

Wireless Sensor Network. 5