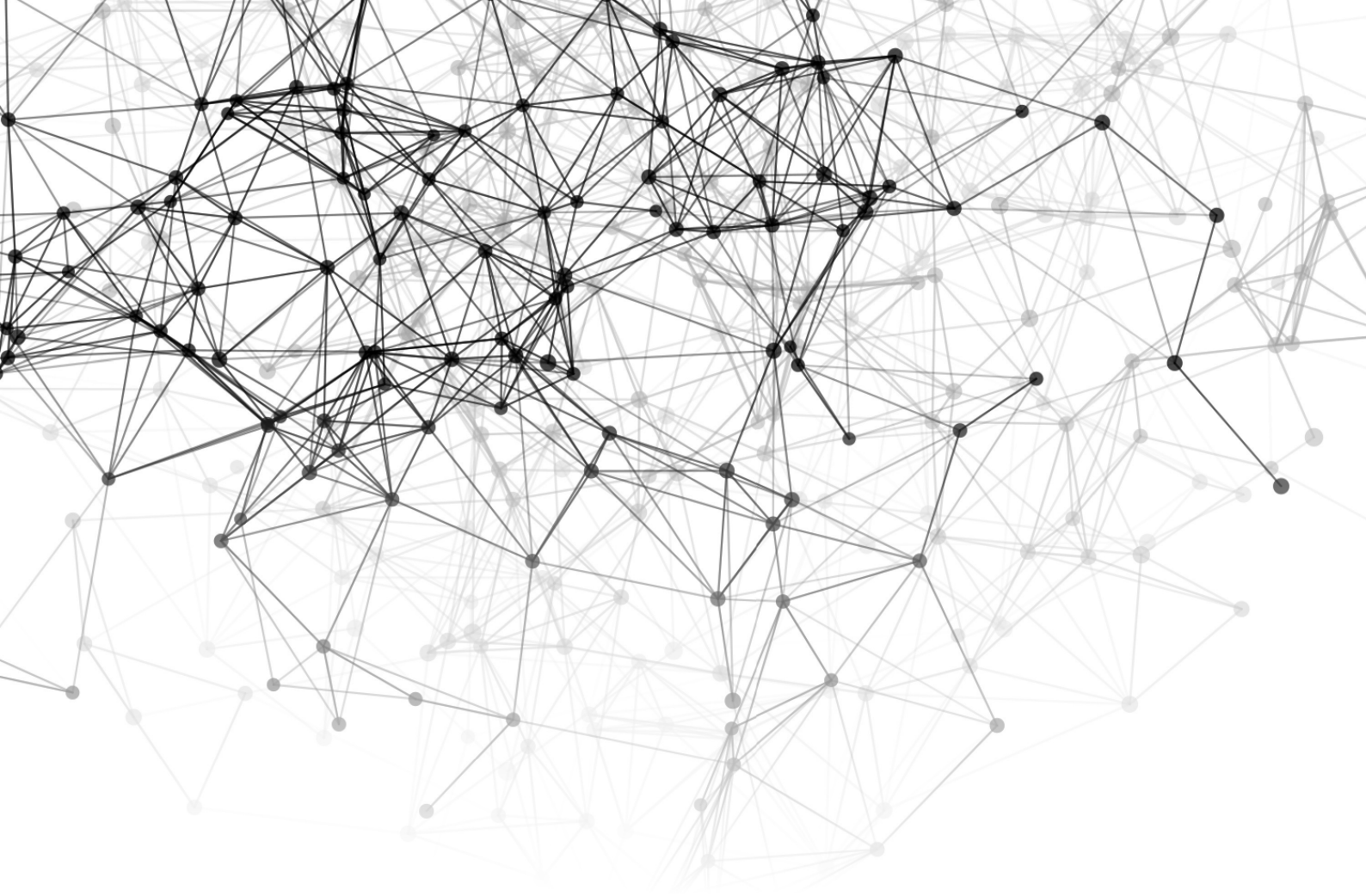


VisuAAL

An Application for Visualizing Realistic Mesh Network Protocol Behavior Through UPPAAL Simulations

Software Master's Thesis
Group des103f17

Kevin Haugaard Jørgensen
Niels Bjørnbak Christoffersen
Rasmus Dan Petersen
Tim Hjermitslev Gjøderum

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Ø
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
VisuAAL: An Application for Visualizing Realistic Mesh Network Protocol Behavior Through UPPAAL Simulations

Department:
Distributed and Embedded Systems

Project period:
Spring semester 2017

Project group:
des103f17

Participants:
Kevin Haugaard Jørgensen
Niels Bjørnbak Christoffersen
Rasmus Dan Petersen
Tim Hjermitslev Gjøderum

Supervisors:
Jiří Srba
Kim Guldstrand Larsen

Signed copies: 6

Pages: 111

Date of completion:
June 2nd, 2017

Abstract:

It can be hard to know the exact behavior of nodes in a self-configuring mesh network protocol, when different topologies and dynamics comes into play. To support this we present the application VisuAAL. VisuAAL uses UPPAAL as a backend to run simulations of wireless protocol models, and helps developers configure, visualize and understand the protocol behavior in dynamic and randomly generated static topologies. We also present the language VisuAAL Query (VQ), which allows developers control over the coloring of nodes and edges in the topology based on arithmetic and boolean expressions using variables in the simulation state.

We have used VisuAAL to simulate mesh networks with hundreds of nodes for a multitude of different protocol specifications and scenarios and show these results. We have also explored techniques to improve the scalability of protocols modeled in UPPAAL.

The companies Neocortec and LinkAiders are working to develop self-configuring mesh network devices that will improve disaster communication. After the end of this project, these companies are planning to use VisuAAL to explore their protocol behavior in a project aimed at disaster relief in the Philippines.

Summary

In a world where IoT becomes more integrated in our every day life, it follows that there will be increasing amounts of wireless devices that needs to be connected. For these wireless devices, we need protocols that facilitates correct and efficient communication of data between nodes. In this project we do not look at creating these protocol ourselves, instead we want to help developers in the domain specific area of distributed networking protocols, to easily gain knowledge and overview of their protocols through visualization.

We created an application that visualizes the behavior of distributed network protocol models. Our application, named VisuAAL, uses UPPAAL as the engine to run simulations and a UPPAAL model of the protocol that is to be observed. We set a number of requirements to the UPPAAL model, that works as an interface. An example of a requirement is prefixes of variables, so that we are able to recognize them, in order to configure and use them for output data.

VisuAAL lets users configure their model, generate dynamic topologies for the model and visualize the simulations created by UPPAAL. It is essential to be able to configure the model, since it allows the user to compare different versions of a protocol. This is useful since protocols may dictate different behavior for nodes under different topologies and topology sizes. Through VisuAAL, we can generate random topologies and inject them in the model.

In the real world, nodes move and thus the connectivity between nodes can change. We support this behavior by being able to dynamically change topologies and other variables during simulations. This we do by updating variables in the model at specific time steps. This functionality we also use to load location and connectivity data from real devices, and use this in the model when simulating the protocol model.

When we have a configured model, we run simulations with UPPAAL and use the output to visualize the protocol model behavior. VisuAAL visualizes the topology with nodes and edges from the simulations. On this topology different types of data can be shown, based on the chosen output variables. The user can with the query language VisuAAL Query (VQ), color nodes and edges based on output variables of the model. The VQ language, allows users to define how data is presented for the nodes and edges and can for instance be used to visualize abnormalities.

We did a number of case studies in the project, to see what and how we can visualize different protocol models. One protocol we tried in our application was the LMAC protocol which is a MAC protocol that utilizes frames. For the LMAC protocol model developed by Fehnker, Hoesel, and Mader [12], we visualized the state of the nodes and showed how the number of frames affect the protocol model's behavior. AODVv2 is an ad hoc routing protocol. Höfner and McIver [23] developed a model of this protocol which we visualized. We focused on the queue of messages that nodes have. If the queue over time becomes large, there is too much traffic for the node and it is unable to keep up. We also did two case studies of two protocol models that we made in [29] based on our understanding of Neocortec's MAC [27] and Routing [26] protocols.

In this project we also explored techniques to improve run time scalability of protocol models, since this can be a obstacle for comprehensive protocols or large topologies.

Preface

This report is a product of a master's thesis project by four Software students at Aalborg University, under the Distributed and Embedded Systems department.

It is expected that the reader has knowledge at the same level as a student at the 3rd semester of the master in Software Engineering at Aalborg University in order for the reader to benefit the most from reading this report.

Special thanks to Thomas Steen Halkier and Jens Laage Olsen from Neocortec for collaborating with us in this project. They introduced their protocols for us, and helped us understand them.

Special thanks to Dmitry Ivanov, for being the first active user and giving feedback on VisuAAL. Also a special thanks to Ansgar Fehnker, Peter Höfner and Rob van Glabbeek for help in understanding the protocols we use and feedback on some of the results from our case studies.

Reading Guide

All figures in the report have been created by the authors, unless otherwise specified. The bibliography can be found at the end of this report on page 113. Appendices starts on page 119. Bibliographical remarks can be found on page 111.

Appendix F presents an overview of the licensinc of our work, including licensing information about the libraries we use. In Appendix ZIP the structure for the attached files can be found.

The format of a printed report is ill suited for displaying simulation animations which make up most of our results. To illustrate the simulation results, we will present and discuss snapshots of interesting simulations in the paper, and include a link to a video displaying the simulation. In addition, the serialized simulation files will be available in Appendix ZIP, such that the simulation can be loaded and run in VisuAAL. In addition, we will include videos of all simulations, where QR codes can be scanned to access the video.

Contents

1 Introduction

Computer systems have revolutionized many aspects of the world. From their use in large scale server parks to mobile phones to small embedded devices, they are a big part of everyday lives. There seems to be a trend where more and more things can be accessed through the internet from social profiles to toasters or other smart devices. This latter part is an example of what we call the Internet of Things (IoT).

The emergence of IoT technology allows computer systems to interact with humans in a number of new intriguing ways. IoT devices are often dependent on wireless communication, because it offers them a large degree of freedom in how they fit in with their environment, but there are also a number of challenges associated with this. As developers we want our software to function in a predictable and correct way, and as users we want the software we use to perform as we expect it to. This is also true for the communication protocols that facilitate wireless communication. Usually, some sort of centralized communication scheme is used, where a central controller facilitates the communication, but it is also possible to have networks without a controller, where the nodes of the network are self-configuring and self-maintaining, known as mesh networks. We deal with mesh networks in this project and will expand on what defines such a network later in this report.

As mentioned, we depend on wireless communication, IoT technology and mesh networks in a myriad of ways, and thus it is important that the protocols and applications that use these are correct and perform according to the specifications set forth for them. Testing large networks of wireless devices running in parallel is a complicated task, partly because large networks inherently contain many individual devices and occupy a large physical space, and partly because collecting data from these many devices is difficult without affecting the connectivity between devices or altering the performance in other ways. One alternative to testing in this context, is the use of statistical model checking (SMC) [31] with a modeled network to estimate the probability of system properties. Formal verification could be used for the same purpose, but due to state space explosion [32] of such models this does not scale to large systems. SMC and simulation can be used to gather performance metrics for mesh network protocols and hereby aid in the further development and improvements of the protocols.

In this project we present VisuAAL, an application that helps visualizing network behavior based on data extracted from the model checking tool UPPAAL [30]. An example of what our application can do is shown in Figure 1.1, where the results from 50 simulations are merged and shown together. Edges between nodes are colored based on how often they were used: Lighter edges are used less than darker edges. Completely red edges are used by all 50 simulations.

The purpose of VisuAAL is to help developers in the domain of mesh network protocols to understand how the performance of their protocol is affected by different parameterizations and alterations. With an existing UPPAAL model of a protocol, the user should be able to retrieve useful data of the protocol, and be able to compare the behavior of different configurations and protocols. VisuAAL will allow the user to configure, setup, run simulations and extract the results of any mesh network protocol model, as long as it fulfills certain requirements.

Different uses of a mesh network protocol might have different areas of interest, thus it is important that users can configure what to visualize and how to show the data. An example where this is important is energy consumption; choosing a good configuration is a trade off

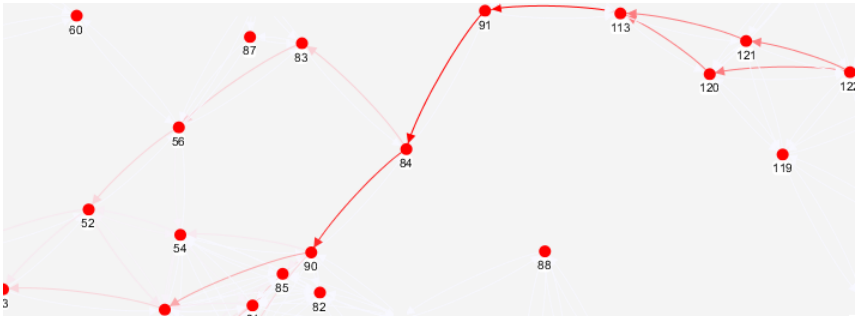


Figure 1.1: Example result from VisuAAL. Edges between nodes are colored based on their usage: The more red an edge appears, the more often it is used.

between energy consumption and data throughput. Thus, a developer should be able to configure the system such that both of these parameters are available, and the correct decision can be made. By adding another layer of information on top of what is given from UPPAAL, it should be possible for the users to easily visualize the behavior of their protocol. In addition, topologies and environments in the real world change over time. These things we support in VisuAAL.

1.1 Problem Definition

In [29], we experienced how difficult it can be to relate the results retrieved from UPPAAL directly to performance data, especially in terms of comparing the results of tests and experiments with real devices with this data. This is particularly problematic for users in the domain of mesh network protocols who are not UPPAAL experts. This is the main problem we aim to solve in this project, where we will work with adding an additional visualization layer on top of UPPAAL, resulting in the behavior being represented graphically, based on a selected topology.

From meetings with the company Neocortec [36], who work with mesh network protocols, we know that there are multiple aspects that are important in an application like the one we want to make, for it to provide them with value. Neocortec would like to be able to spot undesired behavior and have enough data about the situation to be able to understand why a given anomaly occurred. As a way of finding the undesired behavior they proposed that it would be very interesting if we can generate random topologies and run simulations of those. Optimally, anomalies should automatically be found and reported based on some specific predicate. If a topology is found which produces anomalies, it should be possible to make many runs on this topology with different configurations to analyze why and when the behavior occurs. Thus, both running the same configuration on many topologies and many different configurations on the same topology are interesting prospects.

Currently, Neocortec is able to visualize how a network behaves by running their protocol in real time, but changing the topology to an arbitrary topology is difficult and time consuming. We want to at least match this functionality, and to do so we will need to visualize data and map it onto a visualization of a topology. If anomalies are found, it should be possible to extract specific simulations and visualize them. Additionally it should be possible to visualize many simulations simultaneously, assuming they have the same topology.

When Neocortec tests their protocol, they often use a network topology consisting of 64

nodes. Neocortec says that although possible, emulating networks with more nodes becomes cumbersome and expensive in hardware, and thus doing large scale testing for hundreds or thousands of nodes is infeasible. This is where simulation is an extremely viable alternative. In [29], we tested the scalability of two UPPAAL models inspired by Neocortec's mesh network protocols, and concluded that they do not scale linearly with the number of nodes. Because we want to be able to simulate wireless protocols with a high number of nodes in UPPAAL models, we will also need to look into different modeling approaches which could improve the scalability of UPPAAL models. While this is not directly related to VisuAAL, it is still an important subject when we want to perform large scale experiments.

In [29], we show that it is possible to generate queries for UPPAAL models of mesh network protocols, and when these queries are executed they result in sufficient data to visualize the state changes of a mesh network protocol in real time. In this project we continue this work, and to automate this visualization process through VisuAAL. Additionally, we found that running models with many different configurations is cumbersome in UPPAAL, with one configurable aspect being the topology. Neocortec has a desire for testing with many different topologies, and therefore we will look into generating topologies. We will configure models with these topologies, which can be used to obtain a number of simulations that can then be compared.

Another use-case that Neocortec mentioned is that they want to be able to test how a configuration works for a given customer. The customer may have special requirements, or may have some idea of the topology in their system. VisuAAL should then be able to visualize and give the customer some idea of how the protocol is going to work in their concrete use-case. This may include dynamic topologies with nodes changing their locations and the connectivity being dynamic. Specifically, Neocortec will be recording GPS log data as preparation for a project in the Philippines. This project is performed in collaboration with LinkAiders [33], who are developing devices to help with disaster relief as a mesh network capable reporting device. In this context, both Neocortec and LinkAiders are interested in visualizing how the connectivity changes over time, as the nodes move around, and what this means for the protocol performance.

There is an significant challenge in how to allow customization of the visualization while keeping enough generality that different properties may be visualized. We need to accept that there will be some loss of generalness compared to model checking in general, since we will be working in the domain of mesh wireless network protocols where a central aspect of the simulations are the interactions between the nodes in a topology. The topology is of special interest, in that it is central to visualize how the nodes are connected.

In the following we summarize the challenges and ideas that we will look into in this project:

- Design an application domain specific tool for visualization of mesh network protocols
  - Parameterize model variables to examine the impact of different configurations
  - Visualizing UPPAAL results on topologies for one or multiple simulations
  - Generating realistic but random topologies
  - Simulating behavior with dynamic topologies
  - Finding anomalies based on many simulations
  - Loading topology information from external GPS logs
- Examine and improve scalability of wireless mesh network protocol models

1.2 Related Work

In this project we touch upon multiple areas of research. In this section we will discuss these and go through some of the most important related work in each category.

We work with simulation of networking protocols via UPPAAL SMC. A number of protocols have been modeled in UPPAAL [12, 13, 14, 23]. In [12, 13, 14, 23], UPPAAL models are used to check safety properties, and the results of this are mostly represented as graphs, either extracted directly from the graph-builder in UPPAAL or through some other graphing tool applied to the same data. Using this approach to further examine the behavior of the protocols over time is difficult, and mapping this data representation to a network topology is difficult. Our work differs from this, in that we work with visualization directly in a topology centered representation, to provide an overview that more closely represents the reality being modeled.

Van Hoesel and Havinga [52] work with simulation of three different MAC and Routing protocols to examine the lifetime of communication networks on a large scale. Where the other papers we have talked about have used UPPAAL to examine protocols, Van Hoesel and Havinga [52] use OMNet++ [38] to implement the protocols and examine the behavior of random topologies consisting of 46 nodes with various operational parameters, including the message density. Alnuaimi, Shuaib, and Jawhar [2] present a Simulink [35] model of the Zigbee protocol, modeling the physical and MAC layers from this protocol, and use these models to simulate the performance of this protocol under various operational parameters. Our work differs from both of these approaches, in that we use UPPAAL models as the backend for our simulations. By using UPPAAL, we allow the user a formalism that can be used in both VisuAAL and standard UPPAAL for further exploration.

Our work also deals with much larger network topologies than what we have seen previously. When the amount of data becomes very large, visualization becomes a big issue, as it becomes difficult to get an overview of the network. Applications such as DISCO [15] and Gephi [16] provide elegant visualization of large amounts of data, and are thus also related to our work in this regard, but not specialized for network protocol experimentation, and are purely for visualizing data.

2 Mesh Protocol Theory

As mentioned in Chapter 1, we will be working exclusively with mesh networks in this project. In this chapter we explore what defines a mesh network protocol, as well as the related concepts of graph representation, dynamic networks and parametrization.

Chapter Organization This chapter is organized as follows:

- In Section 2.1 we introduce the notion of mesh networks,
- in Section 2.2 we present a way to represent wireless networks as graphs,
- in Section 2.3 we define the terms of parametrization and observation in relation to mesh network protocols.

2.1 Three Categories of Network Topologies

There are different types of networks, which exhibit different characteristics in regards to how the networks operate and which opportunities they provide. Baran [3] proposes three categories which can describe these different topologies.

Centralized is a network where all nodes are connected to a central node which will then handle or forward all communication.

Decentralized network contains nodes in a hierarchic structure where the work normally done by the central node in a centralized network is shared amongst several nodes.

Distributed in this form of network all nodes are connected with all other nodes in its range, thus possibly providing more than one path to each node.

The categories are exemplified in Figure 2.1, where the term Station is equal to what we call Node, and Links represent the edges between nodes. We will use these terms interchangeably throughout this section.

The three categories of topologies differ in how much the individual nodes know about the rest of the network. In a Centralized and Decentralized network, it can be easier to schedule when to send data in order to reduce the risk of collisions because some nodes have a lot of knowledge about the network, which can be exploited to have master nodes that dictate the transmission schedules of their connected nodes, typically through some hierarchical structure based on the connectivity of nodes. This is not the case for a Distributed network, where each node only has knowledge of its first hand neighbors. Because nodes in distributed networks dictate their own transmission schedule in collaboration with their first hand neighbors they are fault tolerant if a node suddenly stops working, e.g. runs out of power. In a distributed network another route through other neighbors can typically be found, whereas the same is not always possible with the centralized or decentralized approach since a single faulty node or link can disconnect the network.

In this report we will deal exclusively with distributed networks, and use the term “Mesh network” interchangeably with this.

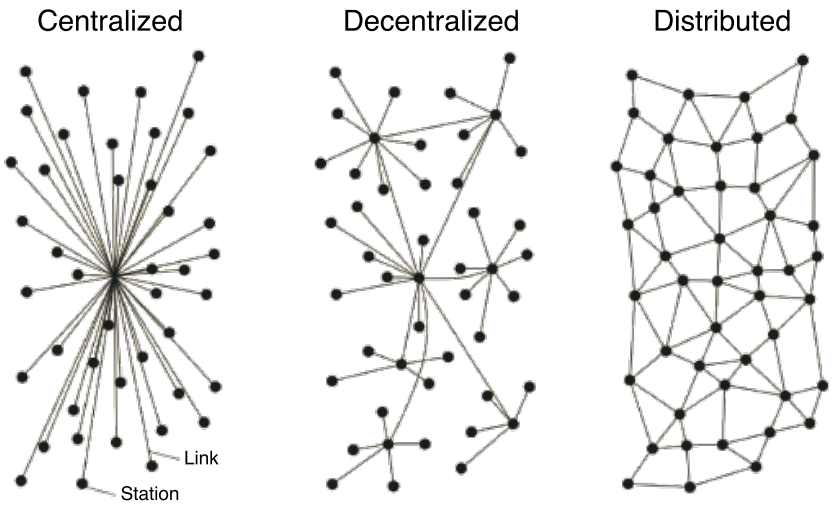


Figure 2.1: Visualization of three different approaches to communication protocols. Original by Baran [3].

2.2 Graph Model of a Network Topology

In order to model the sensor networks we want to work with, we need to be able to make a number of abstractions over the complex real life scenarios we wish to model. The relationship between nodes in a wireless network can be naturally modelled as a graph $\langle V, E \rangle$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges in the graph. In this case the vertices each represent a node, and two vertices v_1 and v_2 are connected by an edge if the transmitter of v_1 can send to the receiver of v_2. The reader should note that only nodes that contain a transmitter can have outgoing edges, and only nodes that contain a receiver can have incoming edges.

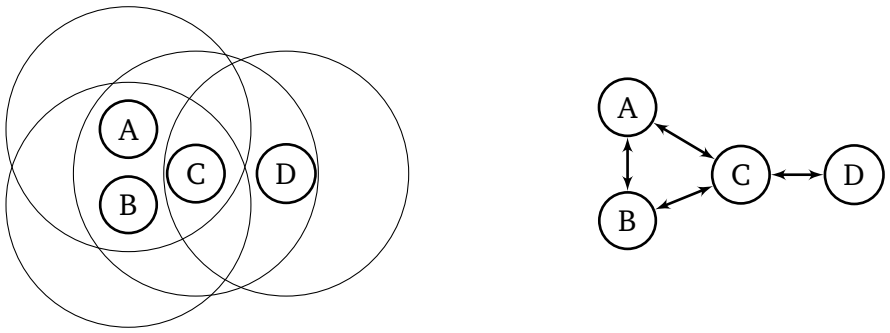
The edges in such a graph are assigned weights which correspond to some relevant aspect of the communication in the network, e.g. the transmission speed or the reliability of a specific communication link. Which aspect is used for the weights on edges depends on which aspect of the network performance is being modeled.

The topology of a network is the specific configuration of nodes and wireless links that are available in the network. The network topology illustrated in Figure 2.2a, shows how a wireless network of 4 nodes and the ranges of their transmitters can be represented by the graph shown in Figure 2.2b.

2.2.1 Dynamic Networks

Until now we have only discussed networks where the nodes and edges have been static. In reality, however, many factors affect the network graph, and many of these can change. Thus we need to consider what it means when a network is dynamic.

When we have wireless communication, the nodes can often move around in physical space. In regards to the graph representation, the physical location does not matter, but when the node moves, new nodes may come into range or remove themselves from range of a node, which affects the connectivity as described in Section 2.2. In these cases, new edges may appear, or



(a) Simple network with 4 nodes and their ranges

(b) Graph corresponding to network

Figure 2.2: Example network and graph describing it

be removed, and the weight of an edge may also change. In addition, in a general network graph nodes may be introduced or removed entirely, representing how new nodes may join or leave the network over time.

In general, modeling dynamic networks as graphs is straightforward, and many mesh network protocols are specifically designed for use in dynamic environments.

2.3 Protocol Parametrization

When we deploy mesh networking protocols, we want the protocol to have the best possible performance. Performance in this context can be many things, for instance power consumption, message throughput, reliability and other factors. Which factors are used depends on the goal of the protocol. The performance of mesh networking protocols is often dependent on the environment they are deployed in and the network they are meant to operate under. For instance, a protocol that provides excellent performance in a low node density network will not necessarily perform equally good in a high node density network, where the chances of collisions may be much higher. In order to allow developers to tune the protocol to the context, most protocols have a number of configuration parameters, which change the operation of the protocol, and thus the performance obtained. These parameters can for instance be the number of time slots in the Slotted ALOHA protocol or the number of re-transmissions to perform before giving up on a given edge.

Selecting the correct parameters for a given use case is often highly dependent on developer experience, since there are an enormous number of environmental factors that need to be taken into account. Predicting the expected performance is thus difficult, but at the same time may have a huge impact on whether a network performs in a fashion that is acceptable to the user.

When we enter the realm of modeling mesh networking protocols, the configuration parameters are also important, because the goal of simulation is often to find how the configuration parameters affects the performance of the protocol, and to predict the performance that can be expected. In modeling, however, the developer is also responsible for modeling the environment, and thus there are also parameters to set that relate to the environment in which the modeled protocol will run. This can include and affect many things, depending on the abstraction level. A few examples are: The topology, physical location of the nodes and changing connectivity

caused by traffic.

In summary, parametrization is important in the context of mesh network protocols, and must be handled in VisuAAL.

In addition, in order to understand how a mesh network protocol behaves, there are certain measurable aspects that describe the observable behavior. When we change the configuration parameters of the protocol, we are looking at the observable behavior to understand how the change affected the protocol performance, and when we examine protocols in different environments, it is the observable behavior that defines the performance.

When we examine networks, we need to take the goal into account, because what is important is defined by the goal. Thus selecting observable phenomenon is highly dependent on the protocol in question, as well as the goal of the particular network. Examples of observable phenomena are: The slot chosen in a time division scheme or the amount of time a node is idle.

3 Timed Automata & UPPAAL Theory

In this chapter we explore the theoretical background for the modeling related work we will perform in this project. Specifically we explore the background for Timed Automata, UPPAAL and how these concepts are applied in the field of modeling communication protocols.

Chapter Organization This chapter is organized as follows:

- In Section 3.1 we provide the definitions for UPPAAL that we will need in this project,
- in Section 3.2 we present Statistical Model Checking (SMC) in UPPAAL,
- in Section 3.3 we summarize the requirements in order to use SMC features,
- in Section 3.4 we describe how we can use UPPAAL to simulate protocols,
- in Section 3.5 we present a simple model of the Slotted ALOHA protocol.

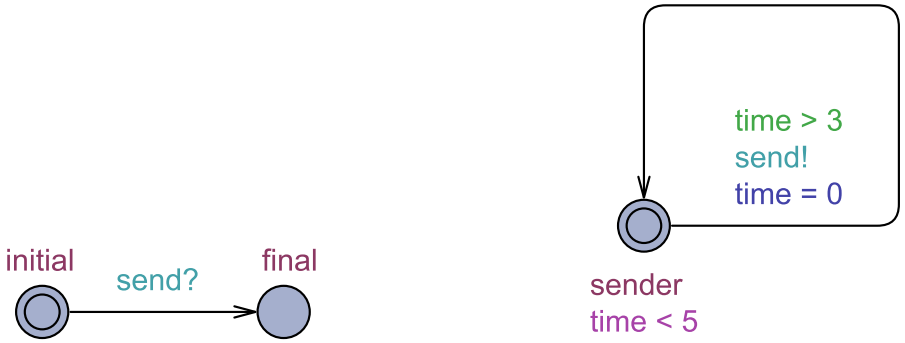
3.1 UPPAAL Introduction

UPPAAL [9] is a tool used to create, reason about and verify models using the formalism of Timed Automata (TA). A common example is to verify whether a model is deadlock free. In this section we take a look at a simple UPPAAL model and afterwards how to verify properties for the model.

To understand the UPPAAL models that will be used later in this project, we need a basic understanding of the parts that make up a model. A UPPAAL model is described as a network of TA, where each TA is described as an instantiation of a *Template*, and a network of TA is thus the parallel composition of these instantiations. The precise definition of a TA is out of scope for this report, but has been described in detail in other papers [5, 9, 30].

In this report we will not go into detail about the theoretical details of UPPAAL, but will rather take a more pragmatic view of the application of UPPAAL as a modeling tool. In order to do so, we need to introduce the notation used, which we will do through a simple model with 2 templates. Figure 3.1a presents one of these. It contains 2 locations with the names `initial` and `final`. The initial location in a template is identified by the double circle. The two locations are connected by an edge, which can be taken when synchronization is initiated on the `send` channel. `send?` indicates that synchronization must be initiated by another process, and that the edge can only be taken as the result of synchronization on the given channel. `send!` indicates that synchronization is initiated when this edge is taken.

The second template from the model is shown in Figure 3.1b. This template has only a single location, `sender`, with a looping edge. Note that this edge has a guard, which ensures that the edge is only enabled when the `time` clock is greater than 3. Additionally the location has an invariant, meaning that we can only be in the location when the invariant holds. In this example the invariant ensures that `time` cannot become larger or equal to 5. Because of the invariant and guard, the synchronization with the `send` channel will happen between 3 and 5 time units. When the edge is taken, the clock `time` is reset to 0, and the loop starts over.



(a) Template 1. Simple template with 2 locations and 1 edge.

(b) Template 2. Template that initiates synchronization.

Figure 3.1: The pink label is a location invariant, dark red labels are location names, the light blue label is a synchronization, the dark blue label is an update, and the green label is a guard.

If we instantiate a Process 1 from Template 1 and a Process 2 from Template 2, the processes will do a channel synchronization between 3 and 5 time units. Process 1 will then move to the `final` location. Note that because we use broadcast semantics, Process 2 will be able to continue to take the looping edge at a point in time between 3 and 5 time units, even though there are no processes that can receive the synchronization. Additionally, the broadcast semantics mean that if two processes were instantiated from Template 1, they would both be able to progress to their `final` locations at the same time as result of the same synchronization.

Each template has its own local declarations, which can contain several types of variables and functions that can be used in the model. Clocks, such as the clock `time` used in Template 1 and 2 are examples of such variables, and represent continuous real time information about the delay transitions that happen.

The state of a model describes the valuations of the variables (and clocks) and the active location of each process at a point in time during a model run. The state space of a model is the set of all such states. The semantics of a UPPAAL model is given as a Timed Labeled Transition System (TLTS), where the states are given as (ℓ, v), where ℓ is a location, and v is a valuation of the variables and clocks of the system. From a given state in this TLTS there are two types of transitions, delays transitions and discrete transitions. A delay transition d increases all the clocks in the system by d time units. A discrete transition has an action a that represents either an input synchronization ?, output synchronization ! or internal transition and leads to another location, while possibly changing variables, depending on the updates attached to the edge, where the action was attached. Following the two transitions types defined:

$$(\ell, v) \xrightarrow{d} (\ell, v')$$

$$(\ell, v) \xrightarrow{a} (\ell', v')$$

UPPAAL uses continuous time, and stores it in a finite number of zones [4]. A zone describes a range of clock valuations, where the set of possible actions in the system is the same. For a process instantiated from Template 1 in Figure 3.1b there are two zones, one where $\texttt{time} \leq 3$ and one where $3 < \texttt{time} < 5$.

3.2 Statistical Model Checking

In newer versions of UPPAAL, a statistical model checker is included, called UPPAAL SMC [6]. This extension makes it possible for us to do statistical queries based on simulations and use these to estimate the probability that a query is fulfilled within a certain timebound, based on a number of simulations run on the system. Applied to network protocols, this could for instance be used to estimate the probability of a collision happening within 100 time units. UPPAAL SMC estimates this probability by generating stochastic runs through the system. The number of these runs that fulfill a given predicate versus the number of runs that do not, can be used to estimate the probability that the predicate is true with a certain confidence.

The concept of generation can be briefly exemplified through the model from Figure 3.1a and Figure 3.1b, where the two templates will select a possible delay. Template 1 has no invariant in the `initial` location, meaning that it will select a delay from a exponential distribution such that it will favor shorter delays. Template 2 has an invariant, and thus is unable to delay beyond a certain bound. It will thus select from a uniform distribution up to the bound imposed by the invariant, unless other is specified. Whichever template has chosen the shortest delay will decide, that all processes will do this delay, whereupon an available action will be performed. The exact method by which runs are generated, as well as the semantics have been explained in depth by David et al. [7].

Statistical model checking is good at quickly estimating the probability of some predicate being true without exploring the entire state space. This allows queries to run on bigger models compared to exhaustive verification where the entire state space is explored, because only a single trace through the system is considered at a time in SMC. This alleviates many of the problems with state space explosion commonly experienced with traditional model checking.

UPPAAL SMC also allows probabilistic weights on edges, which indicate a probabilistic choice between a number of edges. Figure 3.2 shows a small UPPAAL SMC example which has weighted edges. The probability of selecting the edge to location A is 20% whereas the probability of going to location B is 80%. These probabilities are very useful when we are modeling wireless network protocols, because there is always a probability for a message being lost because of the nature of wireless communication.

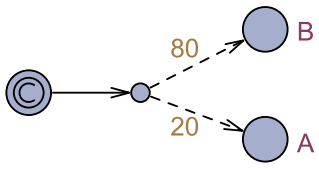


Figure 3.2: UPPAAL SMC example using probability weights on edges

Probabilistic queries use a new syntax, which is described in depth by David et al. [7]. A simple example is provided below, where φ is a predicate and N is an upper bound of how many time units are explored:

Pr [<= N] φ What is the probability that φ is true within N time units?

A concrete example of queries could be the following, using the model from Figure 3.2. Note that the bound is set to 1 time unit. Because the initial location is committed, the edge must be taken without any delay, and thus we can guarantee that the transition has occurred after 1 time unit. We obtained the following results from running the queries:

`Pr [ <= 1] (<> Process.A)` Result: [0.192, 0.212] with 95% confidence

`Pr [ <= 1] (<> Process.B)` Result: [0.789, 0.809] with 95% confidence

[0.789, 0.809] indicates that the (unknown) probability is estimated to be between 78.9% and 80.9% with 95% confidence. Note that UPPAAL estimates the probability based on a number of runs, and therefore cannot provide exact probabilities. For both queries we use a confidence of 95%, which is the default setting in UPPAAL. As expected, we can see that there is around 80% probability of ending in location B and around 20% for location A. It is possible to alter a number of parameters in UPPAAL, and thus affect the wanted confidence and other variables [9].

In conclusion, using UPPAAL SMC is beneficial since it can provide an estimate of the probability of a predicate, making it more practical for use with large models.

3.3 Requirements for UPPAAL SMC Models

UPPAAL SMC imposes three main requirements to models compared to a regular UPPAAL model which are relevant for this project:

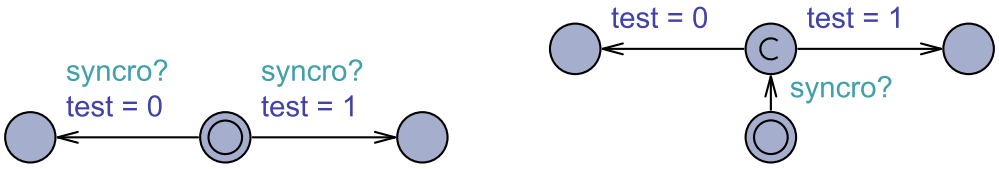
- SMC models must exhibit *Input Determinism*
- SMC models must exhibit *Independent Progress*
- *Priority* between channels or processes is disallowed in UPPAAL SMC.

3.3.1 Input Determinism

A model exhibits input determinism if it describes a deterministic Timed Automaton (TA). David et al. [7] discusses Priced Timed Automata (PTA), which are TA where it is possible to have different clock rates in different locations. Thus, a regular TA is a PTA, where the rates are the same in all locations. We simplify and formulate David et al. [7]'s definition of deterministic PTA as: For all locations ℓ and input synchronizations i in an input deterministic model, the location ℓ' obtained after receiving any input synchronization must be the same for all edges originating in ℓ, that is:

$$\text{if } (\ell, v) \xrightarrow{i} (\ell_1, v_1) \text{ and } (\ell, v) \xrightarrow{i} (\ell_2, v_2), \text{ then } \ell_1 = \ell_2 \text{ and } v_1 = v_2$$

To exemplify this, Figure 3.3a illustrates part of a model that violates input determinism in two different ways. Suppose there is a template that will output on the `syncro` channel at some point in time. When this happens, the model can take one of two different edges. This violates input determinism because the two edges lead to two different resulting locations, either the left or the right. In addition, the two edges result in a different valuation, because the variable `test` will have either the value 0 or 1. In this case, there is a trivial fix, which is shown in Figure 3.3b, where an extra committed location is inserted. The synchronization will always result in moving to this new location, and the updates are retained in the original edges. Because the model is constructed such that there is no interleaving with other processes to take into account, this fix works. One must be careful to ensure that this is fulfilled in general.



(a) Template from model that violates input determinism

(b) Template from model that fulfills input determinism

Figure 3.3: Two version of template from model before and after input determinism fix. Note that there is another template in the full model emitting the `synchro` synchronization.

3.3.2 Independent Progress

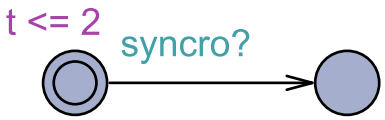
A model exhibits independent progress if it is always possible for a process to delay or do an action without being dependent on other processes. Note that the action can be any output action or the internal transition. David et al. [8] define a model with independent progress as the following: "an implementation cannot ever get stuck in a state where it is up to the environment to induce progress. So in every state there is either an output transition (which is controlled by the implementation) or an ability to delay until an output is possible. Otherwise a state can delay indefinitely. An implementation cannot wait for an input from the environment without letting time pass.". This has the implication that if a location has an invariant, we must ensure that there always exists a way for the template itself to leave the location before this invariant is broken.

Formally, we can say that for any state (ℓ_0, v_0) of the model and any delay $d \in \mathbb{R}_{\geq 0}$, there must exist a sequence of alternating delays and output actions or internal transitions (here referred to as $a_1 \dots a_{n-1}$) of the form

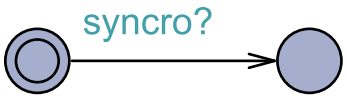
$$(\ell_0, v_0) \xrightarrow{d_1} (\ell_0, v_1) \xrightarrow{a_1} (\ell_1, v_2) \xrightarrow{d_2} \dots \xrightarrow{a_{n-1}} (\ell_{n-1}, v_{2n-2}) \xrightarrow{d_n} (\ell_{n-1}, v_{2n-1}), \text{ such that } \sum_{i=1}^{n} d_i = d$$

To exemplify this, we show two templates in Figure 3.4, one which violates independent progress and one which does not, both with same behavior. Again, assume that there is a template emitting on the `syncro` channel, and that the shown template synchronizes exactly at two model time units. For the first of the two models, UPPAAL will refuse to execute queries against it, since it sees no way to ensure that the `syncro` synchronization occurs within the 2 model time units allotted for it.

We know of no general way of transforming a model to ensure independent progress, but in many cases it is solved by not mixing clocks in invariants and an external template that initiates a synchronization. Often the invariant can be removed, since the synchronization of initiating template forces the receiver to progress. If no synchronization comes, and no invariant is present, the receiver can simply delay indefinitely.



(a) Template from model that violates independent progress



(b) Template from model that does *not* violate independent progress

Figure 3.4: A template (a) that violates independent progress and a template (b) with same behavior as (a) but does not violate independent progress, assuming another template initiates synchronization at time 2

3.3.3 Priority Relationships between Processes and Channels

In classic UPPAAL it is possible to set priorities between channels and between processes [50]. When applied to channels this means that if two edges are enabled with channels a and b being on each edge respectively, the channel with the highest priority will be fired. In the same way, processes may be prioritized, such that a higher priority process will progress before lower priority ones. This is disallowed in UPPAAL SMC.

3.4 Simulation Queries

The inclusion of statistical model checking in UPPAAL allows for simulation queries which we will use extensively in VisuAAL.

In UPPAAL a simulation is a single run of the model. The result of a simulation is a sequence of data points which represents the values over time of chosen variables which are included in the simulation query. In UPPAAL the result of simulations are visualized in a graph.

To illustrate this, we will use the model template from Figure 3.5 and present the data resulting from a simulation query of this template.

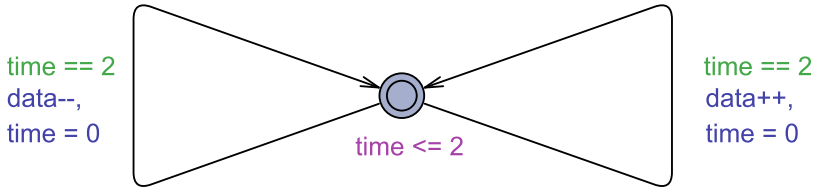


Figure 3.5: Template that manipulates the variable `data` every 2 model time units

The template in Figure 3.5 is a simple template with a single integer variable `data` that is initialized to 10. Every 2 model time unit, the template either increases or decreases `data`. We can then run the following simulation query on the template:

```
simulate 1 [<= 10] { data }
```

If we deconstruct this query, there are three main parts:

- `simulate 1` means that we run a single simulation of the model
- `[<=10]` indicates that the simulation is timebound to 10 model time units
- `{ data }` is the list of variables to watch during the simulation

Since this is a single simulation of the model, there can be many different outcomes of the simulation, one of the outcomes can be seen in Table 3.1 and graphically visualized in Figure 3.6.

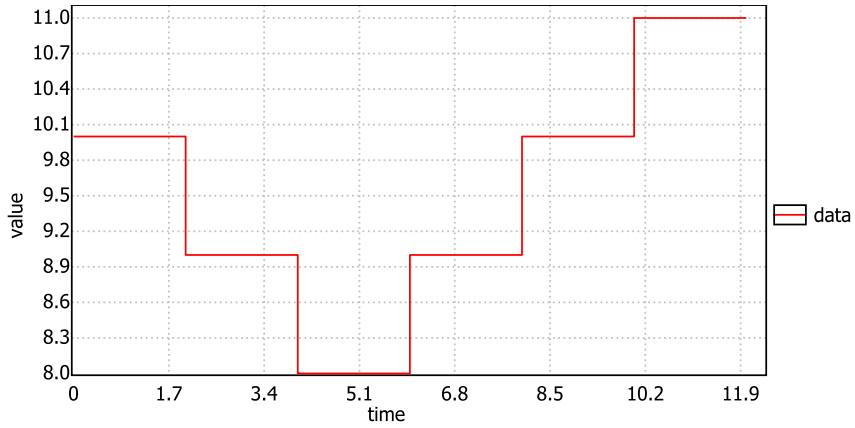


Figure 3.6: Visualized result for simulation query, that is timebound to 10 model time units

Time	0	2	2	4	4	6	6	8	8	10	10	12
Value	10	10	9	9	8	8	9	9	10	10	11	11

Table 3.1: Raw data for simulation query, that is bound to 10 model time units

3.5 Modeling a Simple Protocol

In order to more easily demonstrate how the theory presented in Chapter 2 and Chapter 3 can be used to create a model for a communication protocol, we will now present a simple implementation of the Slotted ALOHA protocol [45]. Slotted ALOHA is a time division based protocol, which facilitates transmission and reception of messages in a wireless network. A single transmission/reception round in this protocol for S slots is described in pseudo code form in Listing 3.1.

```
selected ← randomly select slot ∈{1,2,...,S}
for each slot current ∈{1,2,...,S}:
  if selected = current
    Transmit
  else
    Receive
```

Listing 3.1: Simple pseudo code for Slotted ALOHA

Recall that only one node may transmit at the same time, otherwise collisions will occur. Further notice that in this protocol, the risk of collision is directly linked to the number of slots and neighbors for each node. We have created a simple model which represents this protocol in UPPAAL. This model has two templates, which each have an instantiation for every node. The templates both have a parameter `id`, and can be seen in Figure 3.7 and Figure 3.8.

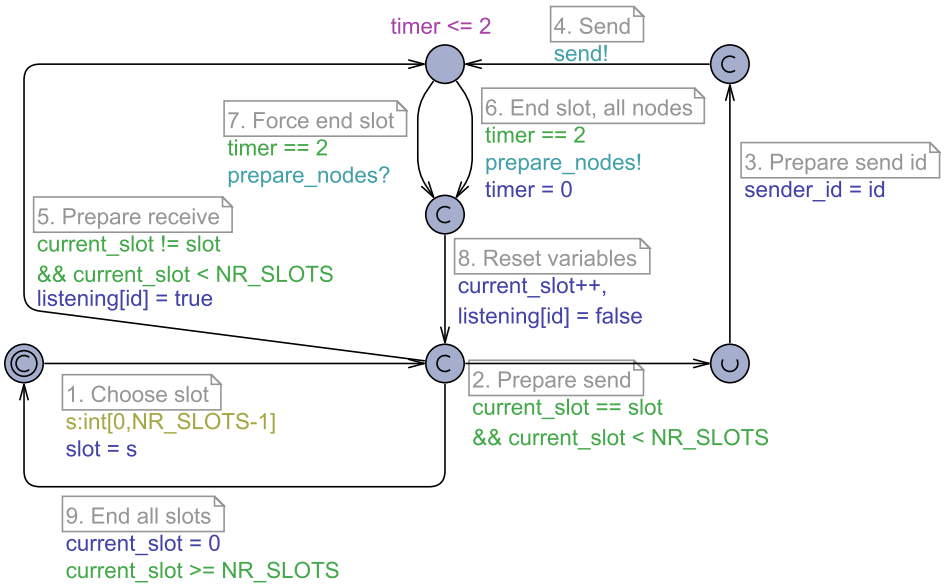


Figure 3.7: Template for Slotted ALOHA sender

The transmission template seen in Figure 3.7 starts by selecting a slot from the committed initial location, and then moves to the top-most location through either the edge annotated with 5 or 2, depending on whether the node has to receive or transmit, respectively. When the time of the slot is over, the sender templates loops back to prepare for the next slot. Note that we use a modeling trick to force all nodes to prepare for the next slot by synchronizing on the `prepare_nodes` channel. If we did not do this, some nodes may still not have left the previous location, and is then not ready to synchronize on the `send` channel as they should. When all slots have passed, all nodes loop back to their initial location and start over.

The receiver template seen in Figure 3.8 is simpler. It start in the bottom-right location, and when a node is to receive a transmission, it moves to the other location. Here, any additional messages in this slot are received, and if more than one neighbor sends in the slot, the collision is recorded.

This model, although simple, provides an overview of the number of messages being sent and lost due to collisions in a network. As a sample of the data that can be extracted, consider

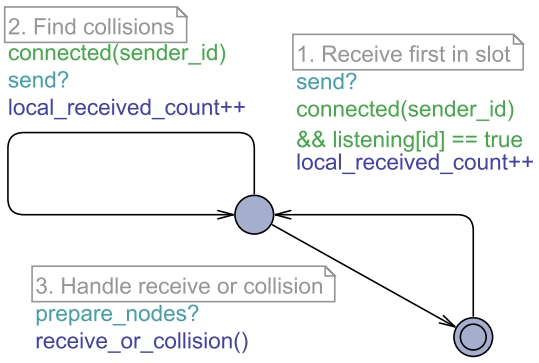


Figure 3.8: Template for Slotted ALOHA receiver

the topology shown in Figure 3.9. This topology can be represented by the incidence matrix where each node is connected to the four closest nodes, such that node 7 is connected to nodes 0, 1, 5 and 6.

To simulate, and thus sample the successful transmission ratio for node 0 in this topology and the protocol configured to 8 slots, we can use the following query in UPPAAL:

```
simulate 1 [<= 1000] {received_count[0], collision_count [0]}
```

When we ran this query, we got the results shown in Figure 3.10, which shows how the number of slots resulting in successful reception and in collisions accumulate over time for node 0. Because of the way the Slotted ALOHA protocol works, there will almost always be some amount of loss, when neighbors choose the same slot. By looking at this data, one could imagine trying to increase the number of slots available in the model, and thus experiment to find an acceptably low amount of lost messages. Recall, however, that this only gives data for a single simulation, and that one must be careful to not compare behavior from a single simulation of each configuration. We will not be experimenting further with the Slotted ALOHA protocol in this chapter.

This example shows how a model for a protocol can be constructed, and how UPPAAL can be used to extract useful information about the behavior of the protocol. This simple model is capable of representing a topology, and represents our understanding of the Slotted ALOHA protocol. We will return to the Slotted ALOHA protocol later in this report.

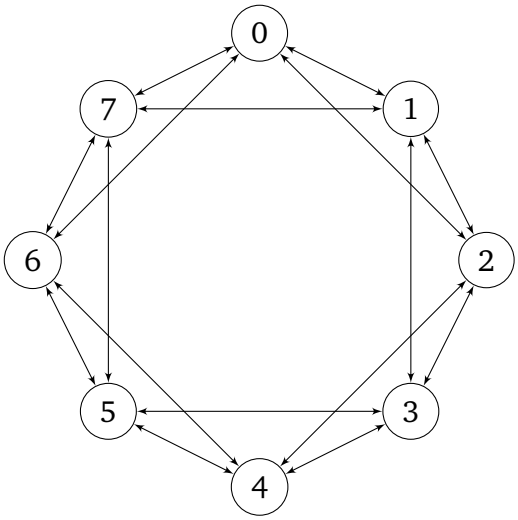


Figure 3.9: Topology for Slotted ALOHA example. Note that all edges are bidirectional.

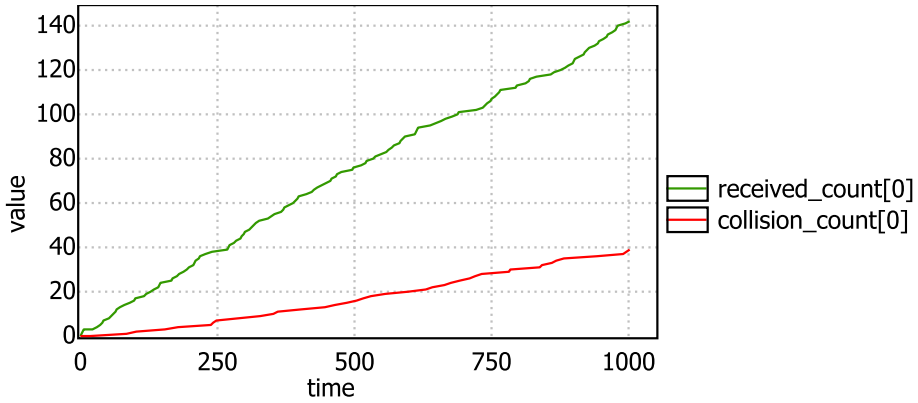


Figure 3.10: Graph for messages received and collisions recorded in Slotted ALOHA protocol model. This graph is extracted from UPPAAL directly.

4 Requirements to Models

In this chapter we will present the requirements for a model to be compatible with VisuAAL and a few tricks to change the model so that channel and location usage may be visualized in VisuAAL. In addition, we continue the Slotted ALOHA example from Section 3.5.

Chapter Organization This chapter is organized as follows:

- In Section 4.1 we describe central concepts to wireless communication protocols which the user needs to encode when modeling,
- in Section 4.2 we describe the general requirements for compatibility between a UPPAAL model and VisuAAL,
- in Section 4.3 we explain how channel and location usage can be enabled for visualization in VisuAAL,
- in Section 4.4 we apply the changes needed to make the Slotted ALOHA protocol model compatible with VisuAAL.

4.1 Modeling Tasks

In the realm of wireless communication protocols there are a number of concepts that are central. These concepts will have to be considered when modeling wireless communication protocols, and thus they are of interest to us as we want to visualize the behavior of such protocols.

In modeling wireless communication protocols there is a distinction between modeling the protocol based on some specification and modeling the environment in which the protocol operates. This distinction also affects the modeling tasks that must be completed in order to end with a model that can provide the answers wanted, and in our case can support visualization. Therefore we will discuss these tasks individually.

In modeling the environment for a wireless communication protocol there are many environmental factors that can be applied, but there are two tasks that are central and necessary for visualization of the protocol; the *topology* and the *number of nodes* for the network that the protocol operates in. The topology represents the directed graph describing which nodes are connected to each other, the number of nodes naturally describes the number of nodes in this graph. In this graph, an edge between nodes a and b exists if a is able to receive messages from b. This graph is typically directed, because usually the range from a node is decided by the transmitter in the node. Because each node has their own transmitter, the ranges of nodes might not be identical, and since the connectivity depends on the range of nodes, the topology graph might not be directed.

The protocols themselves have two main classes of variables that are of relevance for visualization, *configuration variables* and *output variables*. Configuration variables set up the parameters for protocols, and output variables encode the aspects of the protocol that the user

wants to investigate. Output variables are used in queries for UPPAAL to execute simulations that can be used to visualize the behavior of protocols.

4.2 Compatibility Requirements

Ideally VisuAAL should support an arbitrary UPPAAL model of a distributed network protocol. In practice however we need a link between the model itself and VisuAAL. This link we define as a set of requirements for models that VisuAAL supports. By assuming that the models adhere to these requirements, we are more easily able to recognize variable identifiers, which we need in order to configure the model and to generate and run queries for variables.

In this section we list the requirements to model, which models must fulfill to be compatible with VisuAAL.

SMC model As we are using UPPAAL SMC, the model must be compatible with this.

Configuration Variables Variables that should be configurable from VisuAAL should be prefixed with `CONFIG_`. All configuration variables must be instantiated in a single line in their declarations.

Number of Nodes There must be a variable called `CONFIG_NR_NODES`, which describes the number of nodes in the network. This must be a constant variable of integer type. This variable is used for various things such as initializing the correct number of node processes and array sizes.

Topology We need a definition of connectivity between nodes specified as a 2-dimensional int or bool array. The array must be instantiated in a single statement, and it must be named `CONFIG_connected[CONFIG_NR_NODES][CONFIG_NR_NODES]`. All values in the array must be integers, even if the type of the elements is bool. It must be placed in global scope, and not in a function. It must be a non-constant array.

Output Variables Variables used to output data should be prefixed with `OUTPUT_`, so that we can use them in queries. If the output variable is an array, it should be a 1- or 2-dimensional array of size `CONFIG_NR_NODES` in each dimension. It is only possible to use non-constant variables of the built-in types: bool, int, double and clock.

Template Parameter If there are parameterized templates in the model which are not instantiated in the system definition, the parameter must be a bounded type which scales with the number of nodes, so that the number of processes are dependent on the number of nodes, such as `int[1,CONFIG_NR_NODES]`.

Time Conversion Constant It is possible to specify how model time units are converted to milliseconds. If the variable is not declared, we will assume that 1 mtu = 1 ms. This can however be overwritten using the `CONFIG_MODEL_TIME_UNIT` integer constant. As an example, assigning this variable to 20, indicates that 1 mtu = 20 ms.

Since we can alter the topology and thus also the number of nodes through VisuAAL, the model needs to scale with the number of nodes. This makes variable instantiation difficult when the variables are arrays of a size that scales with the number of nodes. For example,

instantiations of arrays of size `CONFIG_NR_NODES` may require the instantiation to be changed as well, because the initialization is specific to a number of nodes. An easy solution to this, is to use a function to instantiate such arrays initially in templates. A special exception to this is the topology, where VisuAAL will handle the configuration when a new topology is needed.

In order to compare different models and configurations, we want to include configuration management. For any interesting mesh protocol model there will be configuration parameters that affect how the model behaves, and changing these parameters can help developers understand how different settings and environments affect the performance, which is one of the key goals of modeling the network protocol.

By only including prefixed variables in VisuAAL, we let the model creator choose which variables that are configurable and create variables for outputting data as well. This means that the user of VisuAAL only needs to know about those chosen variables.

A configuration variable must be instantiated in a single statement. We have chosen this limitation so that we can immediately parse the initial value.

Assuming that these requirements are fulfilled for a model, VisuAAL will be able to visualize the behavior for it.

4.3 Location and Channel Usage as Output Variables

It is not immediately possible in VisuAAL to visualize the locations of templates as we do not include this in a simulation output. If it is desired to visualize when the current state includes a specific location, it is possible to have a boolean variable for the location. For all incoming edges to the location, the boolean variable should be changed to true and all outgoing edges from this location should change the variable to false. If the variable is located in a template with an instantiation for each node, VisuAAL can visualize the variable for each node. An example of how it can be done, can be seen in Figure 4.1.

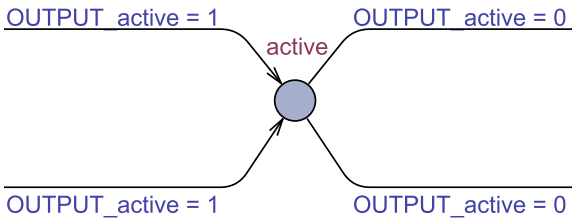


Figure 4.1: Example of how a location can be modeled as an output variable

In regards to visualizing channels, we can follow a similar approach. An example of how it can be done can be seen in Figure 4.2, where the topmost locations are from one template, and the bottommost locations are from another. We assume the model instantiates a process for each node with the parameter `id`, which then communicates over the channel `broadcast`. If we want to simply visualize whether a channel has been used, we create a two dimensional array of size `CONFIG_NR_NODES` in both dimensions, since it will be possible for each node to transmit to each other node. We call this array `OUTPUT_broadcast[][]`.

The template that initiates synchronization with `broadcast!` should somehow transfer its own id to some global variable, in our case we call this `sender_id`. When the receiving nodes perform their update, they will receive the updated `sender_id`, and then mark the proper

index in `OUTPUT_broadcast`. This will log whether the broadcast channel has ever been used between two nodes. If one is interested in resetting the array, this can be done with a committed location from the receiver.

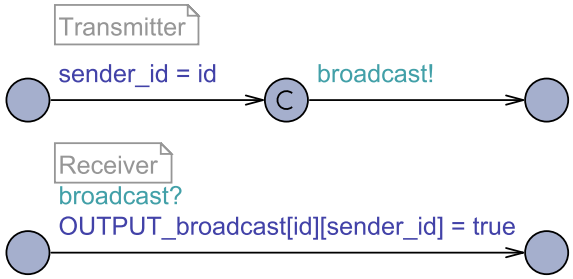


Figure 4.2: Example of how a broadcast channel can be modeled as an output variable

4.4 Requirements Applied to Slotted ALOHA

If we return to the model for the Slotted ALOHA protocol, which was presented in Section 3.5, we can illustrate how this model can be made compatible with VisuAAL. In this simple model, we modeled a topology with 8 nodes, which means that we had a variable `CONFIG_NR_NODES`, as well as a topology definition. The complete global declaration from the model after it has been made compatible can be seen in Listing 4.1.

One requirement we had to fulfill was that the number of templates must scale with the number of nodes. This is handled by the typedef on Line 3 in Listing 4.1, which ensures that the correct number of processes is being instantiated because an `id_t` parameter has been used for the two templates in the model.

```
1  const int CONFIG_NR_NODES = 8;
2  const int CONFIG_NR_SLOTS = 8;
3  typedef int [0, CONFIG_NR_NODES-1] id_t;
4
5  broadcast chan send, prepare_nodes;
6  int OUTPUT_received_count[CONFIG_NR_NODES];
7  int OUTPUT_collision_count[CONFIG_NR_NODES];
8  int sender_id = 0;
9  bool listening[CONFIG_NR_NODES];
10 clock timer;
11
12 int CONFIG_connected[CONFIG_NR_NODES][CONFIG_NR_NODES] = {
13   {0,1,1,0,0,0,1,1},
14   {1,0,1,1,0,0,0,1},
15   {1,1,0,1,1,0,0,0},
16   {0,1,1,0,1,1,0,0},
17   {0,0,1,1,0,1,1,0},
18   {0,0,0,1,1,0,1,1},
19   {1,0,0,0,1,1,0,1},
20   {1,1,0,0,0,1,1,0}
21 };
```

Listing 4.1: Updated global declarations for Slotted ALOHA

We have created two configuration variables, `CONFIG_NR_NODES` and `CONFIG_NR_SLOTS`, where the latter variable defines the number of slots that nodes can choose from. We have renamed the two variables used in Section 3.5, so that they are output variables. The new names are `OUTPUT_received_count` and `OUTPUT_collision_count`, and they maintain a counter for every node for the number of slots that resulted in successfully receiving a message and the number of slots that had a collision in it respectively. Note that these are placed in global scope such that we can index them with the node id. While having this in global scope is not strictly necessary, it enables us to present the variables in a single listing here. They could also have been placed in the receiver template, but as single variables. The access would then have been `receiver(id).OUTPUT_received_count` for instance, instead of `OUTPUT_received_count[id]`. Both these solutions are compatible and possible in VisuAAL.

With regards to the topology, it was already encoded as an int array, and placed in the correct scope, and thus all we had to do was to rename it to `CONFIG_connected` in order for the model to be compatible with VisuAAL.

5 VisuAAL - Implementation Details

As mentioned in Chapter 1, the overall goal of this project is to allow visualization of protocol behavior through UPPAAL and UPPAAL models. In this chapter we will present an overview of VisuAAL, which is a tool aiming to solve this. We will present the interfaces of VisuAAL components, and present the features and some implementation details of VisuAAL.

Chapter Organization This chapter is organized as follows:

- in Section 5.1 we explain our choice of verifyta as UPPAAL executor,
- in Section 5.2 we introduce the components of VisuAAL,
- in Section 5.3 we describe how UPPAAL model files are stored,
- in Section 5.4 we explain special configuration variables and how they can be updated,
- in Section 5.5 we present how we can make dynamic changes to the protocol models,
- in Section 5.6 we present a format of a special log file, used to specify dynamic topologies,
- in Section 5.7 we describe how we parse the output variables,
- in Section 5.8 we go in depth with VisuAAL Query and how this can be used to customize visualization,
- in Section 5.9 we show how we calculate average simulations, that can show data from multiple simulations simultaneously.

5.1 Verifyta

We want to include simulation facilities in VisuAAL. For this we use UPPAAL to run the simulations and provide the results to VisuAAL. We have explored two approaches to this, a command line tool and a Java library. Using the UPPAAL Java libraries in VisuAAL is problematic because of licensing issues. Additionally, the library has dependencies to the user interface from UPPAAL, which is problematic. An alternative is a stand-alone command line tool included in the UPPAAL installation called *verifyta*. Verifyta is used to run queries for a model, and writes the output back to the command line. Verifyta takes two parameters: A model and a file containing one or more queries as input. This allows us to run simulations as an external application, and without requiring the full user interface from UPPAAL. The downside of verifyta is that it returns the output as text, and we will need to parse this output in order to use it in VisuAAL. Since verifyta is an external application, the user would need to point to an existing UPPAAL installation, which also alleviates the licensing issues. For the above reasons we use verifyta in VisuAAL.

5.2 Components in VisuAAL

To use verifyta in VisuAAL, we need a number of components to interface with verifyta and handle the modeled protocol. In Figure 5.1 an overview of these components are illustrated. Note that this is a dependency graph to illustrate the interdependencies of the components in VisuAAL. The following sections briefly explain the responsibilities and requirements for each component.

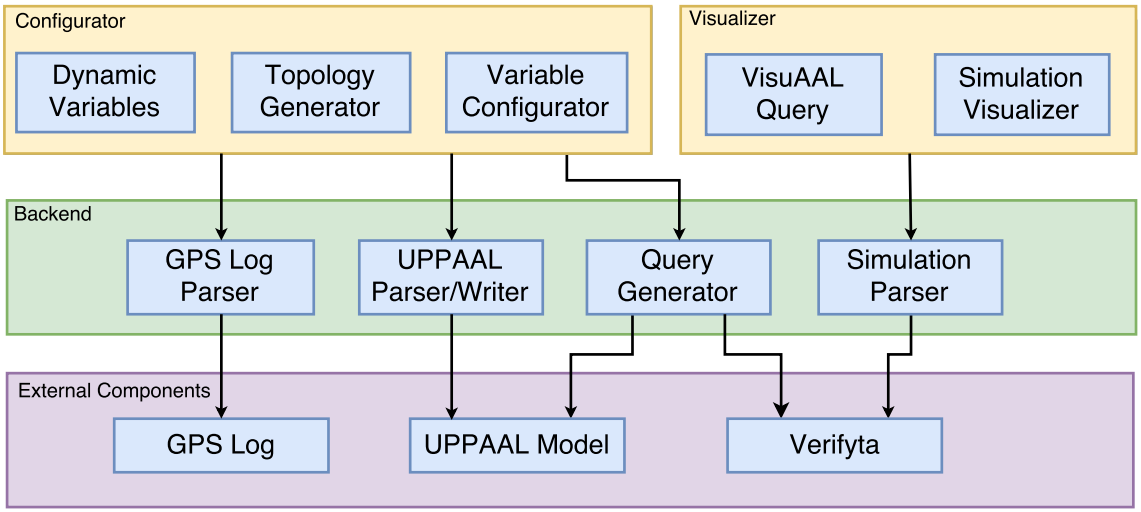


Figure 5.1: Overview of components. The arrows represents dependencies, such that if a component has an arrow to another component, the first is dependent on the other component.

5.2.1 Configurator

The Configurator component is responsible for all the changes that we can do to the UPPAAL model before we execute the queries. This includes modifying configuration variables, altering the topology and dynamic updates. Their responsibilities include:

Dynamic Variables The user is able to add dynamic updates, which represents assignments of variables in the model at specific times. This can be used to alter the behavior of the protocol over time, to test how environmental changes over time affect the network. It could for instance be used to modify the topology over time.

Topology Generator When working in the realm of distributed network, the topology is of high interest. We have therefore created a rather elaborate topology generator. The user pans to a location in Google Maps [19], where the nodes should be added. The area from Google Maps is divided in a grid, where each cell is customizable with the mean and deviation for standard deviations from which the number of nodes and the range in meters on the nodes are drawn. The nodes in each cell are spread according to a uniform distribution, and the connectivity function is constructed according to which nodes are within other nodes' transmission range. After generating a topology, the user can move the nodes around if adjustments are needed.

Variable Configurator The user can update the initial value of the different configuration variables through the program.

5.2.2 Visualizer

The Visualizer component is responsible for presenting the results for the user. To allow a wide range of visualizations, it is possible for the user to describe how the results should be presented. There are two subcomponents in the Visualizer component:

VisuAAL Query (VQ) This component contains a parser and interpretor for the VQ language, which is a small language we created to make it possible for users to specify how and what variables of a protocol model they want visualized. This language will be explained in depth in Section 5.8.

Simulation Visualizer This component visualizes what happened in different simulations. It visualizes the topology and given a time stamp, it will present the values at that time. The edges and nodes in the network is colored based on a VQ expression. The topology is shown as a graph using the library GraphStream [22].

5.2.3 Backend

The Backend component contains multiple subcomponents which read from a UPPAAL model and interpret the details we need, as well as create queries for a model and parse the output to something that can be visualized.

GPS Log Parser Component which parses a special GPS Log file, and generates dynamic updates which changes the topology during the simulations.

UPPAAL Parser/Writer This component can parse and update the declarations in a UPPAAL model. All declarations is written in a C-like language. We use ANTLR4 [40] to parse and update declarations.

Query Generator After the user has configured the model and the topology, this component will generate a query which is used to gather results from UPPAAL. This component will also start the UPPAAL Executor.

Simulation Parser After UPPAAL has calculated simulations, this component will parse the output and provide the data to the visualizer component.

5.2.4 External Components

The external components represents the components which we have not created but only employ.

GPS Log A special log file, which includes a nodes location and neighbors at a given time. The format is described in depth in Section 5.6.

UPPAAL Model This component is the users UPPAAL model in XML format. It should satisfy the requirements that we set forth in Chapter 4.

UPPAAL Executor This is the UPPAAL engine, where we use the verifyta executable. We provide the model and a query to verifyta and it returns the result as text.

5.3 UPPAAL Model File Structure

A UPPAAL model file is stored in XML format and contains declarations, templates, system declarations and optionally queries. In order to configure models, we need to be able to parse and modify UPPAAL model files. Additionally, we will need to parse and write the different declaration blocks, which are written in a C-like language. We use the CFG shown in Appendix A.

As mentioned, we provide verifyta with a model and a query file. Each line in the query file must contain a valid query for the model. Because we use verifyta in VisuAAL, we need to provide such a query file, and to create this we need to know the variables that are needed to get the desired query results. We use the output variables to generate valid simulation queries for verifyta.

5.4 Configuration Variables

For configuring a UPPAAL model, we extract all global and local declarations. UPPAAL models typically have many variables, but typically, only some of them are relevant for configuring the protocol. These we call configuration variables. We filter all instantiations from the model to only consider those that are prefixed with `CONFIG_`.

There are three special configuration variables that have a certain meaning in VisuAAL:

`CONFIG_NR_NODES` This required variable encodes the number of nodes in the topology

`CONFIG_connected` This required array variable must be defined as a two dimensional array of int or bool type which specifies the connectivity between nodes. The indices must be initiated with 0 or 1, even if the array is of bool type.

`CONFIG_MODEL_TIME_UNIT` This is an optional integer configuration parameter, which can be used to specify the relation between model time units and milliseconds. A value of 30 indicates that a single model time unit encodes 30 milliseconds. We do not support having a model time units cover only a fraction of a millisecond. This variable affect the visualization, so that we may display the correct number of milliseconds of real time covered.

5.4.1 Updating Configuration Variables

When the user configures variables in VisuAAL, those changes must be propagated in the UPPAAL model. We do this using a lexer we created with ANTLR4 [40], which parses and changes the declarations block where the changed variable was declared. When we reach the instantiation rule of the updated variable we replace the original value with the updated value. This is also used to configure the model with a given topology.

All changes to the model are made to a temporary file, which is also used when executing queries. This allows the user to experiment with configurations, but retain the original model.

The user can choose to save a configured model, and overwrite the original model if this is wanted.

5.5 Dynamic Variables

For VisuAAL we want to add the functionality that allows the user to update variables at specific time steps during simulations. This could for instance be used to alter the topology over time, to see how the protocol behaves in specific preprogrammed and repeatable scenarios.

We define a variable update as a triple: $\langle time, variable, value \rangle$, representing a specific time to perform the update, the variable name to alter, and the value that should be assigned. Typically, there will be multiple such updates, which we may write for n updates as:

$$\left(\langle time_1, variable_1, value_1 \rangle, \langle time_2, variable_2, value_2 \rangle, \ldots, \langle time_n, variable_n, value_n \rangle\right)$$

To support this, we need to modify the model in question such that there is a process that changes the variables at the given time steps. For the above n updates, this can be accomplished with the template shown in Figure 5.2, which uses the clock `visualizer_updater_clock`. All updates are sorted on time when generating the template. The template is automatically added to the system declaration, such that it is included in the model.

The approach we have chosen comes with the limitation that the template is only able to change global variables, since the variables in other templates are not in scope of the new template.

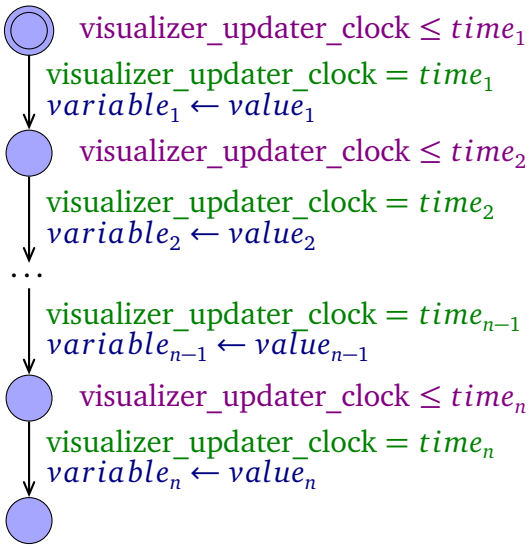


Figure 5.2: Generic template to update variables over time

If the model can make an action at the exact same time as a dynamic variable is set to change, a choice between the two actions is made. An example could be that the dynamic update specifies that an link between two nodes are lost at time 5. If the protocol then checks for this connection at the exact same time, the connection can either exist or not, depending on the interleavings of the actions. Optimally we would always perform the change to the dynamic

variable first, but this is not easily done in UPPAAL, without changing the existing templates, which we want to avoid as it would be complex to manage the templates in a general way. In classical UPPAAL it is possible to add a higher priority to specific processes [50], which would have been the preferable solution. However, as mentioned in Section 3.3 these priorities are not supported in UPPAAL SMC. Due to time constraints we do not look into how this can be solved.

5.6 Creating Dynamic Topologies

During this project, We were approached by Neocortec [37] and LinkAiders [33] regarding their shared project. They would like to to simulate the behavior of their protocols, based on some real-life examples of how the topology could look like. To support this, we have created a format for a log file, which should contain information about which neighbors each node has, and where it was located at that time. The information in the file, can easily be mapped to topology changes in a model and GPS coordinates we can use to visualize in VisuAAL. Using VisuAAL, it is then possible to change the topology during a simulation, based on a log file, using the same structure to alter the model as presented in Section 5.5. Each line in the GPS Log file should be formatted as follows:

```
timestamp; nodeID; gpsLat; gpsLng; neighborID1 RSSI1 neighborID2 RSSI2..
```

The following guidelines apply for the log file:

- Timestamps are represented in milliseconds
- The lowest time stamp will be used as a start time, and will be subtracted from all other log entries
- All node IDs are integers, starting from 0 without skipping a number
- The neighbor list can be empty
- `;` should be used as data separator
- Newlines are used to separate log entries
- Whitespace is used to separate neighbor node IDs and RSSI
- All neighbor node ids should be followed by their corresponding RSSI
- RSSI can be a negative or positive integer
- If the same neighbor is present multiple times in the neighbor list, the latest RSSI will be used
- Lines starting with `//` are ignored

Following is an example of a line from such a log file. At time 28 node 9 has the GPS location (57.0112724, 9.9959301) and has the neighbors with IDs 5, 2 and 11 with RSSIs -40, -54 and -68 respectively:

```
28; 9; 57.0112724; 9.9959301; 5 -40 2 -54 11 -68
```

Based on the neighbor list, it is possible to generate dynamic updates of neighbor connections in the network topology. Additionally when presenting results which uses a GPS log, we also change the result view, to update the topology accordingly and also to move the nodes to the locations specified in the log file.

5.7 Output Variables

In order to visualize the behavior of communication protocols, we need to execute a UPPAAL query. By default, all output variables are included in queries for simulations, so that the user will be able to choose between them when visualizing model simulations. When we parse output variables, the valid output variables must fulfill the following 3 requirements:

- Types of output variables must be int, bool, double or clock
- Variable names must have `OUTPUT_` prefix
- Output variables must be non-constant

The most restrictive of these requirements is that the type must be one of the built-in types, since creating records from these types is a common practice in UPPAAL modeling. We have chosen to limit ourselves from records types and only support the built in types, in order to spend more time on other aspects of VisuAAL. In addition, every use case that we are aware of for a record type output variable can be emulated by placing local variables directly either in global scope or local scope.

We require a prefix, since we want to provide the user with a list of possible output variables and allow them to choose which variables to track, and we do not want to provide the user with all possible variables in the model. We have chosen a simple prefix, `OUTPUT_`, as this is easy to understand, does not limit expressiveness and is easy to parse.

All output variables must be non-constant, since constant variables never change during the simulation, visualizing these over time is pointless.

5.7.1 Output Variable Arrays

We allow the use of arrays as output variables, to signify variables that are bound to either node or edge behavior. Such variables may be placed in two different scopes: Local template scope or global scope. They are defined in the following manner, based on the scope:

Node Variables Variables bound to nodes

Template Scope Single variable

Global Scope 1-Dimensional array of size `CONFIG_NR_NODES`

Edge Variables Variables bound to edges

Template Scope 1-Dimensional array of size `CONFIG_NR_NODES`

Global Scope 2-Dimensional array of size `CONFIG_NR_NODES` in both dimensions

If an output variable is a 1-dimensional array and is placed in global scope we require the array to have exactly `CONFIG_NR_NODES` elements, so that there is an index for each node. If a global array is a 2-dimensional array, we require each dimension to have `CONFIG_NR_NODES` elements, such that there is an output variable for each potential edge in the network. In this case, the first index represents the source node of the edge, and the second index represents the destination node.

If an output variable array is placed in local scope, it must be a 1-dimensional array, and it must have `CONFIG_NR_NODES` elements. In this case, each index is an edge from the node itself to the node of the index.

Arrays of any other dimensionality, of any other type or with any other number of elements will not be considered as valid output variables.

If an array is selected as output variable, all elements are added to the query, even though not all indexes might be used, during a simulation. We cannot know beforehand which indexes are needed.

5.8 VisuAAL Query

In order to let the user define how edges and nodes are visualized, we create the *VisuAAL Query* (VQ) language. The reason we create a language for this, is that it is a very generic way for the user to define a node and edge visualization based on a set of variables with values. We choose to visualize behavior for node and edges in VisuAAL using coloring.

The user is able to define a mapping from values to colors, either as a gradient or specific colors. For gradients, we let the user set a minimum and maximum bound, and a color for each bound. The user then provides an arithmetic or boolean expression involving 1 or more output variables. This expression is evaluated to a number value, and based on this value, a gradient color is chosen to color either edges or nodes based on the user input. When the user maps specific colors to values, a number of colors and values for those are specified in addition to a default color that all other values map to.

This can be used to visualize the states in simulations. It could as an example be used to indicate the state of a network at a glance, for instance idle nodes marked gray or the connectivity of edges being marked with a gradient color of red to indicate how reliable the connection is.

5.8.1 VisuAAL Query - Syntax

A VQ expression contains two essential parts; a mapping between colors and values and an expression that can be evaluated and mapped to a color.

We will present the syntax in Extended Backus-Naur form (EBNF), and the entire syntax for VQ is also available in ANTLR4 format in Appendix C. Precedence of VQ expression operators are standard, such that multiplication having higher precedence than addition for instance. The full precedence can be seen in Appendix C. To improve readability of the syntax in this report, we use $\langle\ \rangle$ to denote optional rules, as we use [and] in our syntax for literals. Recall that { } denotes zero or more repetitions in EBNF.

The color literal used in the syntax, includes the built in colors used by the library GraphStream [21], with only a few examples are shown in the syntax. The expression e of the VQ syntax includes a number of operators, real number constants c, boolean constants b and variables x. ID is an identifier mapping to a real number value. This will be further defined in the semantics.

Following the syntax is shown:

$$
\begin{aligned}
vq &::= gradient\ e \mid colors\ e \mid e \\
gradient &::= [\ \text{color}\ \langle : \ c \rangle\ ,\ \text{color}\ \langle : \ c \rangle\ ] \\
colors &::= [\ \text{color}\ :\ c\ ,\ \{\text{color}\ :\ c\ ,\}\ \text{color}\ :\ *\ ] \\
e &::= (e) \mid e_1\ \text{op}\ e_2 \mid -e \mid !e \mid e_1\ ?\ e_2\ :\ e_3 \mid c \mid b \mid x \\
\text{color} &\in \{\text{red},\ \text{green},\ \text{blue}, \ldots\} \\
\text{op} &\in \{*,\ /,\ +,\ -,\ <,\ <=,\ >,\ >=,\ ==,\ !=,\ \&\&,\ ||\} \\
c &\in \mathbb{R} \\
b &\in \{\text{true},\ \text{false}\} \\
x &\in ID
\end{aligned}
$$

We will also show the syntax for coloring through an example. There are two ways to define the colors, which are then followed by an expression e. The first way defines two colors with a minimum and maximum bound for a gradient color as shown in Equation 5.1. Values outside of these bounds will round to the closest bound. In the second way we can define a number of colors that are all mapped to values, in addition to a default color for all other not mapped values, as shown in Equation 5.2.

```
[white:0, black:50] e                    (5.1)
[red:0, blue:1, green:2,black:*] e       (5.2)
```

5.8.2 VisuAAL Query - Semantics

We will now go through the semantics of the VQ language. We do however only formally define the semantics for the expression here, and presents the coloring specification semantics through examples afterwards.

Following we define two functions, for mapping an ID to a real number and evaluating an expression to a real number, where Exp is the set of all expressions:

$$
\begin{aligned}
v &: ID \rightarrow \mathbb{R} \\
[\![\ ]\!] &: Exp \rightarrow \mathbb{R}
\end{aligned}
$$

Next we present how constants and variables are evaluated:

$$
\begin{aligned}
[\![c]\!]_v &= c \\
[\![b]\!]_v &= \begin{cases} 1 & \text{if } b = \text{ true} \\ 0 & \text{if } b = \text{ false} \end{cases} \\
[\![x]\!]_v &= v(x)
\end{aligned}
$$

Following we present the semantics for the expression operations in VQ:

$$
\begin{aligned}
[\![(e)]\!]_v &= [\![e]\!]_v \\
[\![-e]\!]_v &= -[\![e]\!]_v \\
[\![!e]\!]_v &= \begin{cases} 1 & \text{if } [\![e]\!]_v = 0 \\ 0 & \text{otherwise} \end{cases} \\
[\![e_1 * e_2]\!]_v &= [\![e_1]\!]_v \times [\![e_2]\!]_v \\
[\![e_1 / e_2]\!]_v &= [\![e_1]\!]_v \div [\![e_2]\!]_v \\
[\![e_1 + e_2]\!]_v &= [\![e_1]\!]_v + [\![e_2]\!]_v \\
[\![e_1 - e_2]\!]_v &= [\![e_1]\!]_v - [\![e_2]\!]_v \\
[\![e_1 < e_2]\!]_v &= \begin{cases} 1 & \text{if } [\![e_1]\!]_v < [\![e_2]\!]_v \\ 0 & \text{otherwise} \end{cases} \\
[\![e_1 <= e_2]\!]_v &= \begin{cases} 1 & \text{if } [\![e_1]\!]_v \leq [\![e_2]\!]_v \\ 0 & \text{otherwise} \end{cases} \\
[\![e_1 > e_2]\!]_v &= \begin{cases} 1 & \text{if } [\![e_1]\!]_v > [\![e_2]\!]_v \\ 0 & \text{otherwise} \end{cases} \\
[\![e_1 >= e_2]\!]_v &= \begin{cases} 1 & \text{if } [\![e_1]\!]_v \geq [\![e_2]\!]_v \\ 0 & \text{otherwise} \end{cases} \\
[\![e_1 == e_2]\!]_v &= \begin{cases} 1 & \text{if } [\![e_1]\!]_v = [\![e_2]\!]_v \\ 0 & \text{otherwise} \end{cases} \\
[\![e_1 != e_2]\!]_v &= \begin{cases} 1 & \text{if } [\![e_1]\!]_v \neq [\![e_2]\!]_v \\ 0 & \text{otherwise} \end{cases} \\
[\![e_1 \,\&\&\, e_2]\!]_v &= \begin{cases} 1 & \text{if } [\![e_1]\!]_v \neq 0 \wedge [\![e_2]\!]_v \neq 0 \\ 0 & \text{otherwise} \end{cases} \\
[\![e_1 \,||\, e_2]\!]_v &= \begin{cases} 1 & \text{if } [\![e_1]\!]_v \neq 0 \vee [\![e_2]\!]_v \neq 0 \\ 0 & \text{otherwise} \end{cases} \\
[\![e_1 \,?\, e_2 : e_3]\!]_v &= \begin{cases} [\![e_2]\!]_v & \text{if } [\![e_1]\!]_v \neq 0 \\ [\![e_3]\!]_v & \text{otherwise} \end{cases}
\end{aligned}
$$

5.8.3 Visualizer Query - Example

In this section we show an example, where we have a gradient and an expression:

```
[white:0, black:50] x * (y - z) + 20
```

The query sets the minimum bound to 0 with the color white, and the maximum bound to 50 with the color black. There is then a simple arithmetic expression operating on the three variables x, y and z. In Figure 5.3, we construct a syntax tree for the expression of the VQ above,

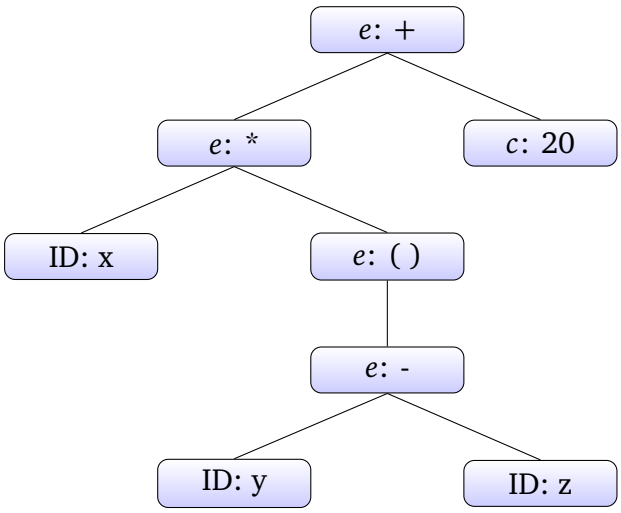


Figure 5.3: Example of a syntax tree for an expression of the VQ language

Figure 5.4: VQ gradient example

where we can see how the precedence puts the parenthesis and its contents at the bottom, so that it will be evaluated before being used by other operators, when the evaluation is done as a depth first tree walk through the syntax tree.

These variables in the expression would be bound to output variables. Suppose x, y and z are output variables that bind to a single node. Then, an evaluation of the expression can be used with the color specification gradient shown in Figure 5.4 to give a shade of gray, which the node will then be colored as.

For the concrete valuation x:4, y:10, z:7, the expression is evaluated as follows:

$$4 \times (10 - 7) + 20 = 32$$

Thus, the color at the 32 mark would be selected for the node, as seen in Figure 5.4.

5.9 Average Values for Simulations

When you run many simulations, it is often beneficial to consider the behavior in each simulation, as well as the average behavior of multiple simulations. It can be hard to get an overview of a protocol's behavior, if it is only possible to visualize one simulation at a time. To solve this, we introduce a special simulation, which contains an average of the data point values for each variable of all simulations and presents it as a single simulation. From this point we refer to this as an *average simulation*.

All the values for data points that we visualize are handled as real numbers, and thus we do not differ between data types of the output variables. We calculate the average simulation by taking the average of the data points, whenever a data point changes, as illustrated in

Figure 5.5. The illustration includes two simulations and the average simulation of these. For the two simulations, the value is different at different points in time. This is generalized, such that each data point is weighted with $1/n$, where n is the number of simulations, and when a data point arises, the weighted value is added to the value for the relevant output variable in the average simulation. This generates an average of the values over time, which encompasses all the simulations generated.

The average simulation is generated just after a simulation is completed by UPPAAL, and thus it is included in the simulation file for later analysis. We only support an average simulation over all simulations, and not for an arbitrary subset of simulations.

The average simulation can be handled as a normal simulation in VisuAAL and can be visualized with the same VQ expressions as the other simulations. Note that since we have a data point whenever the variable changes in any simulation, we will in most cases have more data points in an average simulation compared to a normal simulation, which could prove a bottleneck for updating the GUI for large amounts of data and large topologies. To alleviate this, we collapse data points at the exact same time stamp into one data point, which handles all the data points for that time stamp.

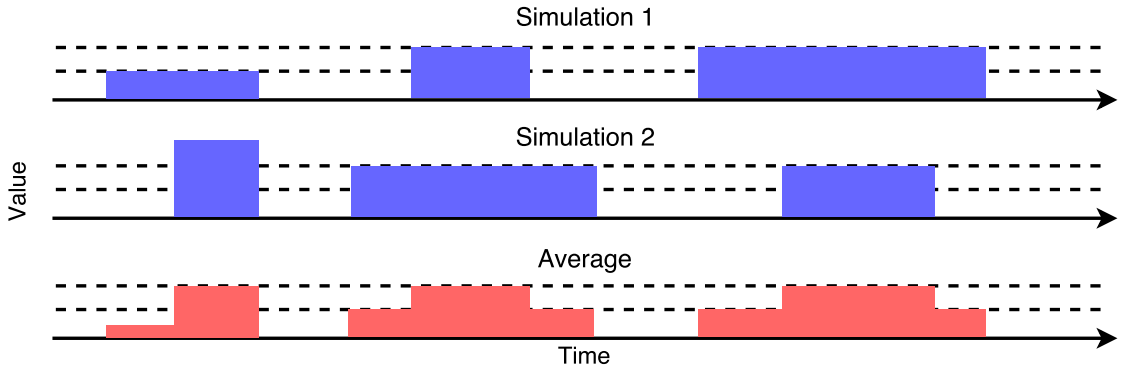


Figure 5.5: Conceptual diagram of simulations 1 and 2, and the average simulation of these

6 Case Study - LMAC Protocol

In this chapter we will briefly describe the Lightweight Medium Access (LMAC) protocol [52]. We will not delve into details in this section, but rather we will give an introduction to the protocol and present a model representing it, and applying VisuAAL to obtain visualization of the behavior of the LMAC protocol. For a more thorough walkthrough of the protocol, we refer to Van Hoesel and Havinga [52].

Chapter Organization This chapter is organized as follows:

- In Section 6.1 we briefly describe the terms used in the LMAC protocol,
- in Section 6.2 we glance over a UPPAAL model representing it, and go through the changes needed to make the model compatible with VisuAAL,
- in Section 6.3 we present the results obtained from applying VisuAAL to the model.

6.1 Protocol Overview

The LMAC protocol was first proposed by Van Hoesel and Havinga [52] and is designed for low power distributed networks; specifically distributed sensor networks.

The LMAC protocol uses Time Division Multiple Access, where time is divided into slots. If nodes in the LMAC protocol select unique slots, they can communicate collision free. In this protocol, certain special nodes act as *gateways*, which are nodes that can receive messages. Other nodes can relay messages, but are unable of being the final destination for messages. Nodes can only select a slot which none of their neighbors have selected, but will otherwise select randomly from a predetermined range of slots. For this to work, all nodes must retain information about their neighbors, and in which slots the neighbors will send their messages. When a node detects that no neighbors are to send and the node itself does not have to send, it sleeps in a low power mode.

There is a chance of nodes selecting the same time slot, which may have been selected by another node during this time. If this happens, the LMAC protocol depends on a third party to inform the two colliding nodes of the situation, after which the colliding nodes will back off for a time dependent on the unique identification number of the node, and then select a random uncontrolled slot.

Van Hoesel and Havinga [52] propose using a *hop-distance* scheme to achieve routing in the LMAC protocol. In such a scheme, nodes retain information about how many hops all their neighbors have to a given gateway in a routing table, and regularly broadcasts its own hop-distance for each gateway, where the number of hops to gateway is one more than the minimum hop distance of any of its neighbors. These broadcasts are called *synchronization signals*. Synchronization signals also include information about the next transmission time to ensure local synchronization of listening and transmission periods, as well as the node id for the transmitting node. When a data package needs to be sent to a gateway, a node will look in its routing table and find the neighbor with the lowest hop-distance toward the gateway

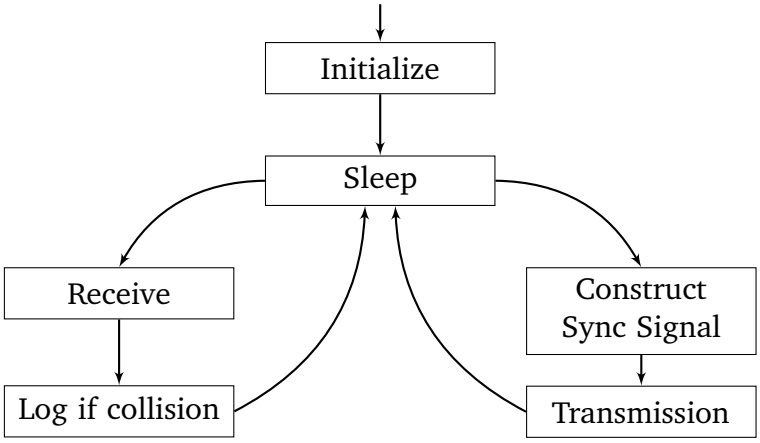


Figure 6.1: Sketch of transitions between the modes of operation for the LMAC protocol

and then constructs an *address package* marking which neighbor was supposed to receive the message.

Gateways are required for the network to be initialized, since when the network starts, the nodes have to be synchronized for this protocol to work. Normal nodes are unable to start up before they have received a synchronization signal from another node. To get the network started, gateways broadcast a message, which is in turn received by the one-hop neighbors of the gateway, which will in turn further transmit this synchronization signal to the two-hop neighbors of the gateway and so forth. In this manner, the network is initialized.

The reception period is split into two main parts, with the latter again being split in two parts. In the first part synchronization signals are received, and in the second part data packages are received, but only if the address package sent before it matches the node id. Otherwise, the node goes to sleep to preserve power.

Transmission is likewise split in two, with the first part dedicated to transmitting a synchronization signal and the second part dedicated to transmitting data packages.

We describe our understanding of the LMAC protocol in three components each handling a mode of operation, and a fourth component being used to construct the synchronization signal. This behavior is described in pseudo code form in Listing 6.1, which loosely represents the LMAC protocol. The transitions between the three different modes of operation and sleep is sketched in Figure 6.1.

```
1  Initialization:
2    Receive synchronization signal
3    Update neighbors with hop distances
4    Construct and broadcast synchronization signal
5    Sleep until time to send or receive
6
7  Transmission:
8    Construct and broadcast synchronization signal
9    g ← Destination gateway for data package
10   n ← Neighbor with lowest hop distance to g
11   Construct and broadcast address package addressed to n
12   Broadcast data package
13   Sleep until time to send or receive
```

```
14
15 Receiving:
16   Receive synchronization package
17   Update neigbors with hop distances
18   Receive address package
19   If collision in slot
20     Log collision in slot
21   If address package = myid
22     Receive data package
23   If syncronization signal indicated collision in this nodes' selected slot
24     While no free slots available
25       Sleep for random backoff period
26       Listen and detect free slots
27       If such a slot exists, randomly select one
28   Sleep until time to send or receive
29
30 Construct synchronization signal:
31   Add node ID to synchronization signal
32   Add Δ time until next transmission
33   for each gateway g in the network:
34     h ← Lowest hop-distance toward g of any neighbor
35     Add h+1 to synchronization signal for g
36   If collision has been logged
37     Add collision slot to synchronization signal
38   Sleep until time to send or receive
```

Listing 6.1: Pseudo code for LMAC protocol

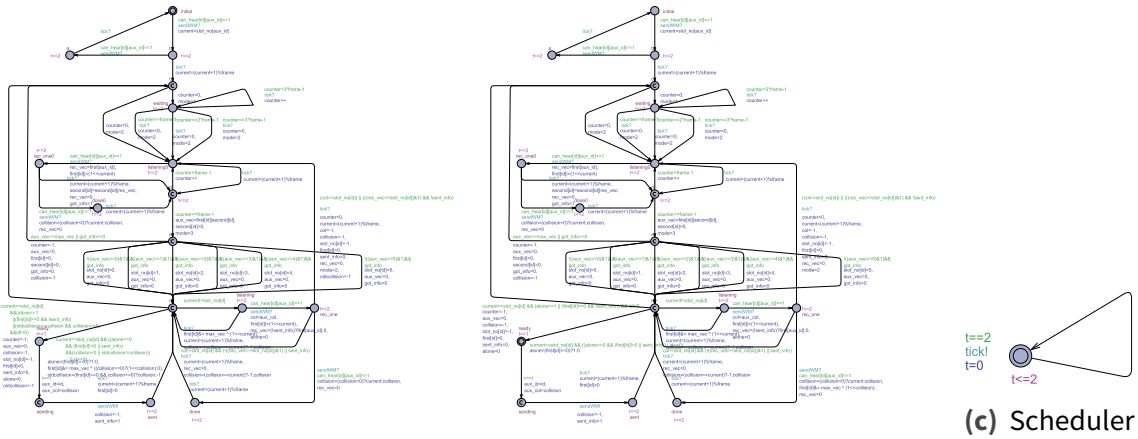
6.2 Model Overview & Compatibility

In this section we will quickly go through the UPPAAL model developed by Fehnker, Hoesel, and Mader [12], with special focus on the changes we had to make in order to make the model compatible with VisuAAL.

The initial model developed by Fehnker, Hoesel, and Mader [12] represents the LMAC protocol, which was briefly sketched in Section 6.1. For a complete explanation of the model, we refer to [12]. The purpose of the protocol model is to explore topologies where there can be unresolved collisions. Recall that nodes select time slots in which to transmit. This may lead to collisions where multiple nodes transmit on the same time slot. One essential feature of the LMAC protocol is that a third party is needed to detect collisions, and it is this feature that Fehnker, Hoesel, and Mader [12] examine. There are some topologies where the LMAC protocol fails to recover from collisions, and the goal for Fehnker, Hoesel, and Mader [12] is to improve the protocol iteratively, such that the number of problematic topologies decreases.

For our case study we have chosen to use the latest version of the model available on Ansgar Fehnkers' webpage [11], model 13. To get a feeling of the size of this model, all the templates can be seen in Figure 6.2. We do not expect these to be readable at the scale used in this report. Note that the gateway and node templates are similar in structure, but differ in the initial locations and variable instantiation values.

Fehnker, Hoesel, and Mader [12] present their results in a tabular form, where the state of each node, including the slot it has selected is visualized with colors and text. This table shows



(a) Node template **(b)** Gateway template **(c)** Scheduler template

Figure 6.2: The original LMAC model developed by Fehnker, Hoesel, and Mader [12]. The main difference between the node and gateway template is that gateway has selected a slot from the very beginning.

how the model progresses over time to illustrate unresolved collisions. Thus, these results can certainly be extracted from the model as it is.

Visualization wise it would be interesting to look into the state of nodes, and the time slot chosen by each node. This is all information that binds to a single node, which changes during normal operation of the protocol model. Additionally, Fehnker, Hoesel, and Mader [12] examines all topologies of sizes 4 and 5 in their paper. A framework that allows automatic topology generation and testing would thus clearly be beneficial for them, although fully enumerating all topologies quickly becomes unmanageable as the number of nodes grows, as they also remark.

6.2.1 LMAC Protocol Model

Before we are able to handle the model in VisuAAL, we need to ensure that it is compatible with UPPAAL SMC. In their paper, Fehnker, Hoesel, and Mader [12] use ordinary UPPAAL syntax to model the protocol, since the goal of the modeling is to do exhaustive search, rather than statistical model checking. This allows the model to be used with ordinary queries as explained in Section 3.1, but we have found that the model is incompatible with UPPAAL SMC, and thus also incompatible with SMC-only queries, such as probabilistic queries and simulation queries.

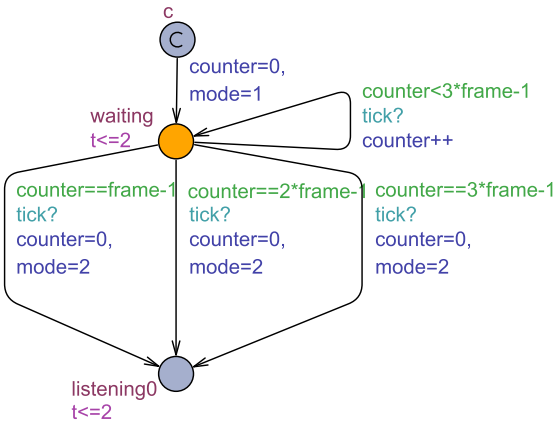
Luckily, the model is compatible with UPPAAL SMC, except for two issues. The two main issues that make this model incompatible with SMC are: lack of input determinism and independent progress.

A minor issue is that the topology in this model is transposed, in relation to our requirement. To solve this, we changed every lookup in the topology, such that instead of accessing `CONFIG_connected[id_1][id_2]`, we access `CONFIG_connected[id_2][id_1]`.

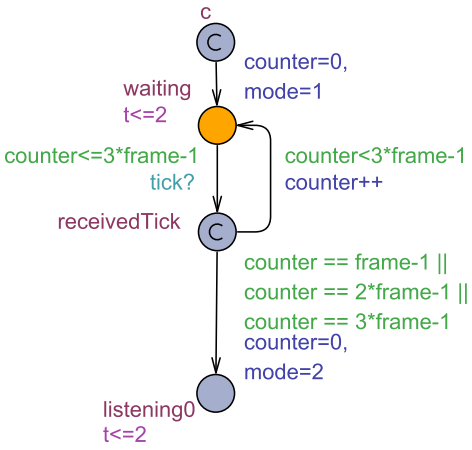
The template gateway is similar to the node template, and thus all the same changes that we make to one template has to be replicated in the other.

6.2.2 Ensuring Input Determinism in the LMAC Model

The requirement of input determinism is violated in the LMAC model in the location `waiting`, as shown in Figure 6.3a, where there are four outgoing edges, which all require the `tick?` synchronization to be taken. `tick!` is broadcast from the scheduler template, which broadcasts every 2 time units. Taking the top right edge will end in the location `waiting`, whereas taking the other three edges will result in the location `listening0`, and the valuations are different as well. UPPAAL SMC will not accept this construction, and thus we need to rewrite it. Luckily, this is fairly trivial with the guards and updates being fairly similar. We have updated this part of the model with the construction shown in Figure 6.3b which shares the same behavior as the original. In this new model, we use a single location, `receivedTick` so that all outgoing edges from `waiting` lead to the same location. A guard is added to the edge, so that `tick?` may only be received if it matches one of the edges where this was possible before, and `receivedTick` is made committed, so that all nodes in this location will have to move out before time can pass, and before any other template may progress, but with no control over the interleaving of when each node does so.



(a) Excerpt from LMAC model. The problematic location is the location named `waiting`, and indicated with orange fill-color.



(b) Excerpt from updated LMAC model. Notice that the `waiting` location is still marked with orange as in Figure 6.3a. The new location is named `receivedTick`.

Figure 6.3: Excerpt from LMAC model. Shows how we fix input determinism.

We want to argue more why that this transformation is valid and that the resulting model performs identically to the original. To do this, we need to ensure that no conflicting interleaving can happen, and that all transitions available before are available after, and vice versa.

Introducing a new location is always dangerous, because it may lead to interleaving issues with other templates. However, in this case there will be no issues since the new location `receivedTick` is a committed location, such that no other nodes may progress before this node exits the location. Recall that the model must process all processes in a committed location before we can do anything else. Thus the only interleavings we have to consider are with other processes in committed locations. Notice that there may be multiple nodes in a committed location as a result of broadcast semantics on the `tick?` synchronization, but no nodes can

be in committed locations before the synchronization. Further notice that only local variables are used in the transformed edges, and that no outgoing synchronization is used on these. In summary, no matter which interleaving with other processes is chosen, no delay transition can occur, and the transformation does not result in a new synchronization and is unable to affect the global variables, and thus the transformation does not introduce any changes that allow the process to affect other processes that did not exist before the transformation.

Having ensured that only the local scope and template behavior can have been affected by the transformation, we now need to ensure that the behavior that was available before the transformation is also available after, and that no new behavior is introduced. By looking at the edges transformed, it is trivial to see that the edges after the transformation provide the same traces from from location `c` to location `listening0` as before, with the caveat that any time the `tick?` synchronization is used, the `receivedTick` location must be visited after the transformation.

Thus, the behavior has been preserved by the transformation, but the resulting model exhibits input determinism, since receiving the `tick?` synchronization in the `waiting` location, will always result in ending in the `receivedTick` location.

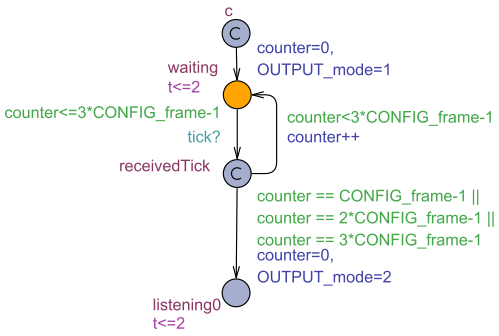
6.2.3 Ensuring Independent Progress in the LMAC Model

We need to ensure independent progress in the LMAC model. We defined this in Section 3.3. As mentioned, a simple scheduler template is used which broadcasts on the `tick!` channel every 2 model time units. Every non-committed location in the templates node and gateway have the invariant `t<=2`, except for the `initial` location and the `ready` location, where the former has no invariant and the latter has `t<=1`. Progress from locations with invariants is ensured since the `tick?` synchronization is ensured every two time units. However, because the synchronization is initiated from another template the requirement for independent progress is not fulfilled. It is simply impossible to end in a situation where a template is stuck in a location with no possible way to progress for more than 2 time units. In addition, if there were, this template would result in a deadlock, and testing for deadlock in the model proves that no such deadlock exists. Despite this, the static code check in UPPAAL is unable to detect that this model will progress, and rejects the model as soon as a SMC query is executed.

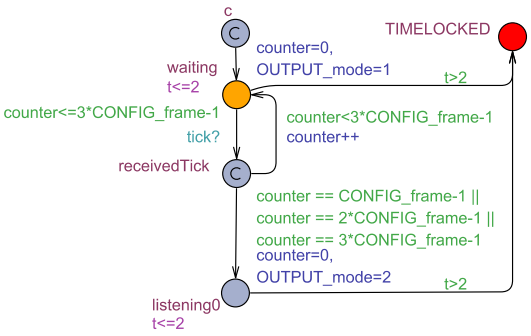
To make the model execute SMC queries, we see two options:

- Either we can rewrite the model such that it is not relying on the external scheduler by moving the clock `t` to local scope in each template, and adding a reset of this clock to every edge that has a `tick?` synchronization.
- Otherwise we could circumvent the static code check of UPPAAL SMC, and allow it to start and run simulations while understanding that if a template ends in a location where no progress can be made, the simulation will break.

We have chosen the second solution because we estimate that introducing a clock in the model for each template will have a larger impact on the performance of the model than the changes needed for the second option. Additionally, we feel that the second option is the least invasive change we can make to the model, and thus also the least likely change to introduce errors in the model. We added an edge with the guard `t>2` to every location with the invariant `t<=2`. This edge will obviously never be enabled, and thus this change is unable to affect the



(a) Excerpt from LMAC model that does not have independent progress. This is the same figure as Figure 6.3b.



(b) Excerpt from updated LMAC model. The red location indicates the new error location.

Figure 6.4: Excerpt from LMAC model, before and after we handle independent progress

behavior of the model but is accepted by the static code checker in UPPAAL. This is exemplified in Figure 6.4b.

6.2.4 Renaming Variables in the LMAC Model

In order for VisuAAL to detect the configuration and output variables of the model we have renamed a number of variables. The changes are summarized in Table 6.1. The variables do not need to be moved to different scopes or changed in any other way than this rename in order for the model to be parsable by VisuAAL. Thus, this does not affect the model functionality in any way.

	Original Variable Name	New Variable Name
Topology	`can_hear[][]`	`CONFIG_connected[][]`
Number of Nodes	`number_of_nodes`	`CONFIG_NR_NODES`
Configuration Variable	`frame`	`CONFIG_frame`
Output Variables	`collisions` `mode`	`OUTPUT_collisions` `OUTPUT_mode`

Table 6.1: Renamings necessary in the LMAC protocol model

6.3 Results for LMAC Protocol

In this section we will describe how we are able to extract valuable information about the performance of the LMAC model through visualization.

The LMAC protocol model has a large focus on discovering which topologies can end in perpetual collisions between nodes [12]. Thus, when we do visualization the main focus will

be on visualizing topologies where there are unresolved collisions. Whereas Fehnker, Hoesel, and Mader [12] performs exhaustive search of all topologies of size 4 and 5, we will instead be doing simulations. This enables us to handle much larger topologies than in the paper with hundreds of nodes, at the cost of only exploring a single trace through the system at a time. This means that we are unable to guarantee topologies as collision free, but we will be able to show large topologies and how nodes keeps colliding.

In order to do so, we will track the `OUTPUT_mode` variable in all gateway and node processes, and thus track the state of each node. The `OUTPUT_mode` variable can have the values shown in Table 6.2, where each value describes the shown mode of operation.

Value	Explanation
0	Node is initializing. No frame has been selected, and node is waiting for initial Synchronization Signal.
1	Node is waiting for a random back-off time before selecting a new frame.
2	Node is listening to detect free frames
3	Node has obtained a frame and is in normal operation mode

Table 6.2: The values and explanations of each mode of operation in the LMAC protocol model. Note that the gateway node starts in mode 3, and that normal nodes start in mode 0.

In order to visualize the state of nodes in a simulated network, we write the following VQ expression:

```
[red:1, blue:2, green:3, black:*] node.OUTPUT_mode + gateway.OUTPUT_mode
```

By summing the `OUTPUT_mode` variables in gateways and nodes we are able to use the same VQ to visualize both these nodes. Note that the first part of the sum will evaluate to 0 for gateway nodes, and the second part will evaluate to 0 for other nodes, since the gateway node IDs and other node IDs are disjoint. Thus, for node ID 0, there will be an instantiated gateway process, but no normal node processes, and vice versa for the other node IDs in the model. When there is no instantiated process with a given name for a given node ID, the referenced variables are evaluated to 0. Thus, the sum in the VQ will evaluate to the mode for a unique node.

As per the pseudo code from Listing 6.1, nodes may select a frame, and then be forced to back off and select a new uncontested frame due to collisions with the neighbors. When this happens, the mode in a node will go from mode 3, through mode 1 and 2 and then select a free frame, if one exists, whereupon mode 3 will be selected again. In our visualization we will see this as color changes from green to red to blue, and back to green if a frame is available. If no frame is available the node will cycle trough mode 1 and 2, until a frame is free. This will be seen as flickering between red and blue.

In our visualization, stable green nodes are nodes that have obtained a frame and which do not collide with their neighbors, and flickering nodes have two causes: either this is a result of nodes not having a free frame available or else it is nodes selecting the same free frame. The

first cause may happen when there are more neighbors to a node than there are frames, such that all the frames are occupied. The second cause may happen for instance if only one frame is available for two connected nodes, and the nodes select the same back-off period. Then the nodes will select the only available frame, which will result in a collision. If this is reported, the nodes will relinquish the frame, and the whole process may repeat.

Both of these situations are interesting, and in our visualization we can see that the first cause will result in red and blue flickering on a node, whereas the second cause will be characterized by flickering between red, blue and green, since the nodes actually end in mode 3 when they select the frame, but will soon be forced to relinquish it.

To run our simulation we generate a random topology. This topology has 126 nodes and is connected. There are areas with high connectivity and places where single nodes connect smaller sub graphs. The topology is shown with the results in Figure 6.6 and Figure 6.8 for the two experiments we will run on the LMAC model.

Experiment 6.1 - Initial Configuration

In the first experiment for the LMAC protocol model we will use a configuration with 6 frames, and see how the protocol behaves in a time bound of 10000 Model Time Units (mtu). Recall that all nodes in the LMAC protocol model progress every 2 mtu, and thus this is equal to 5000 synchronizations on the `tick` channel. We do not know whether this is intended to be translated directly to milliseconds, but in our experience, this is not an unrealistic expectation. The query took close to 4 hours to execute on a standard laptop.

Because there are only 6 frames, there will be situations in the used topology, where there are simply too many nodes competing to obtain a frame, and thus some nodes will be unable to do so. Recall that nodes cannot relay messages across the network before they obtain a valid frame.

An interesting situation can be seen in Figure 6.6, where a number of the nodes are colored blue and red since they are not in possession of a frame. A summary can be seen in Table 6.3. Most of the green nodes are stable, and are thus not in conflict with other nodes. The results are shown in Figure 6.5, or the entire simulation can be seen by loading the simulation file `LMACResult1.sim` from Appendix ZIP.

Figure 6.5: Visualization on the results from Experiment 6.1 which uses 6 frames. Note that the video plays the simulation at 5% of real time, such that it is easier to follow the progression.
The QR Code will redirect to the following link:
`https://youtu.be/FhTTV_cCHEw`

From the video in Figure 6.5, we can see that the initial message is sent to all nodes in the network during the first 10 seconds of the video. If a node stays in a green state, it has successfully obtained a frame. It will try to obtain a new frame, if a collision happens. Furthermore, we can see that many nodes flicker as discussed previously. This is because the number of neighbors in some areas are quite large, thus the 6 available frames are not enough for all nodes to actually obtain a frame. If a node does not obtain a frame, it will never participate in the the network.

# nodes	Node mode	State Explanation
58 %	1: green	Selected a frame
30 %	2: blue	Listening for free frame
12 %	3: red	Waiting random time before selecting a new frame

Table 6.3: Summarization of what is shown in Figure 6.6 at time 4161.3 ms

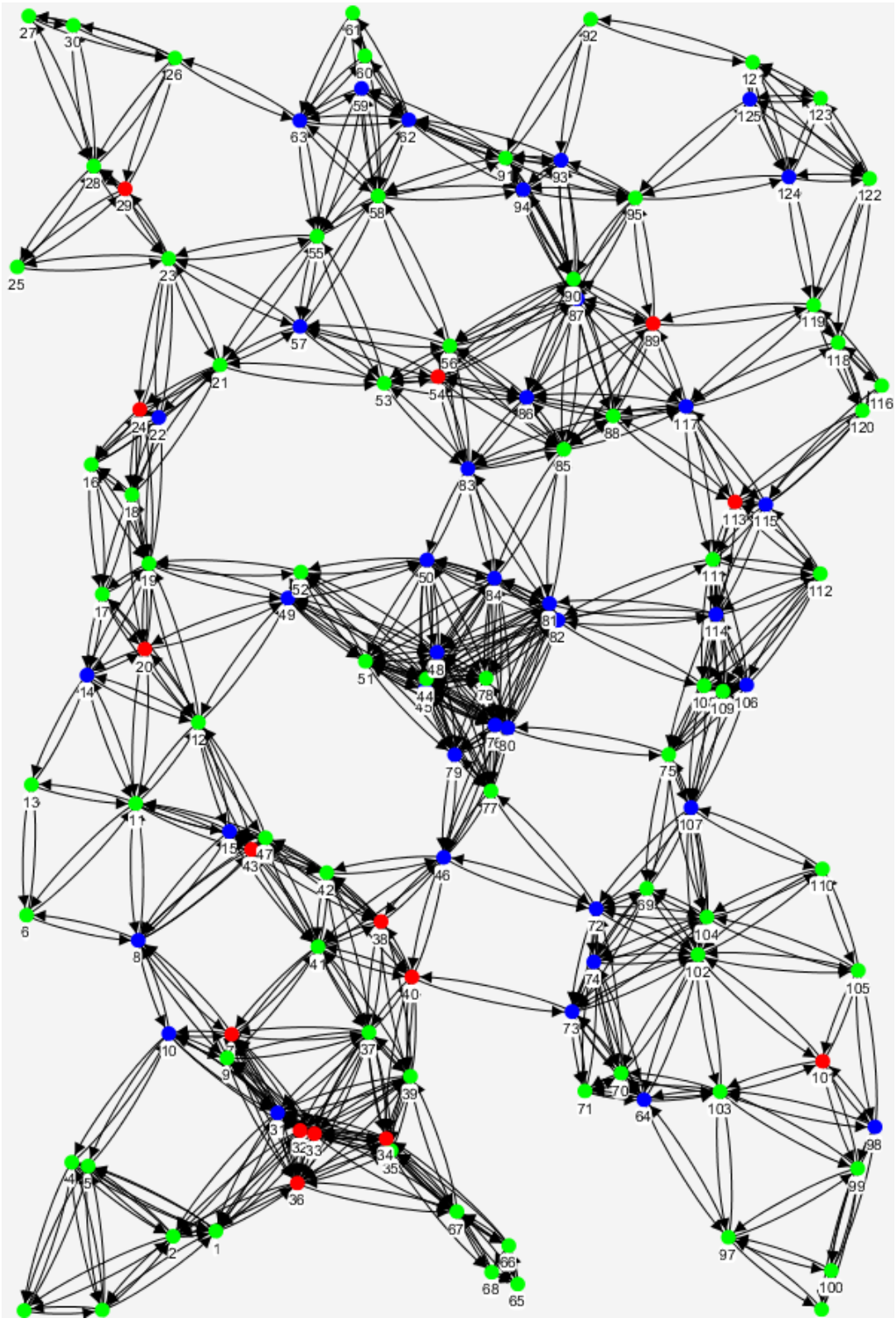


Figure 6.6: Snapshot from first LMAC result experiment. Note that this shows the state at 4161.3 ms. Nodes colored green have successfully obtained an available frame. Nodes colored blue or red, have still not obtain a frame.

Experiment 6.2 - Increased Number of Slots

In order to try and alleviate the problems with nodes not being able to obtain a frame, one might try to increase the number of frames available. We will triple the number of frames to 18, and see how this affects the nodes. We expect to see all nodes obtain frames, and turn green in a visualization rather quickly. This experiment took close to 2 hours to complete on a standard laptop.

The results are shown in Figure 6.7, or the entire simulation can be seen by loading the simulation file `LMACResult2.sim` from Appendix ZIP.

Figure 6.7: Visualization on the results from Experiment 6.2 which uses 18 frames. Note that the video plays the simulation at 5% of real time, such that it is easier to follow the progression.
The QR Code will redirect to the following link:
`https://youtu.be/qRVBK0peEtk`

As expected, the nodes rather quickly obtain a frame. Figure 6.8 shows the state at 1185 ms, where it is clear that all nodes have obtained a frame, and no conflicts are occurring any more, thus all nodes participate in the network. By using visualization, we are able to obtain a clear overview of the behavior of the protocol, and how this is affected by configurations.

A summary of the state shown in Figure 6.8 are shown in Table 6.4.

# nodes	Node mode	State Explanation
100 %	1: green	Selected a frame
0 %	2: blue	Listening for free frame
0 %	3: red	Waiting random time before selecting a new frame

Table 6.4: Summarization of what is shown in Figure 6.8 at time 1185 ms

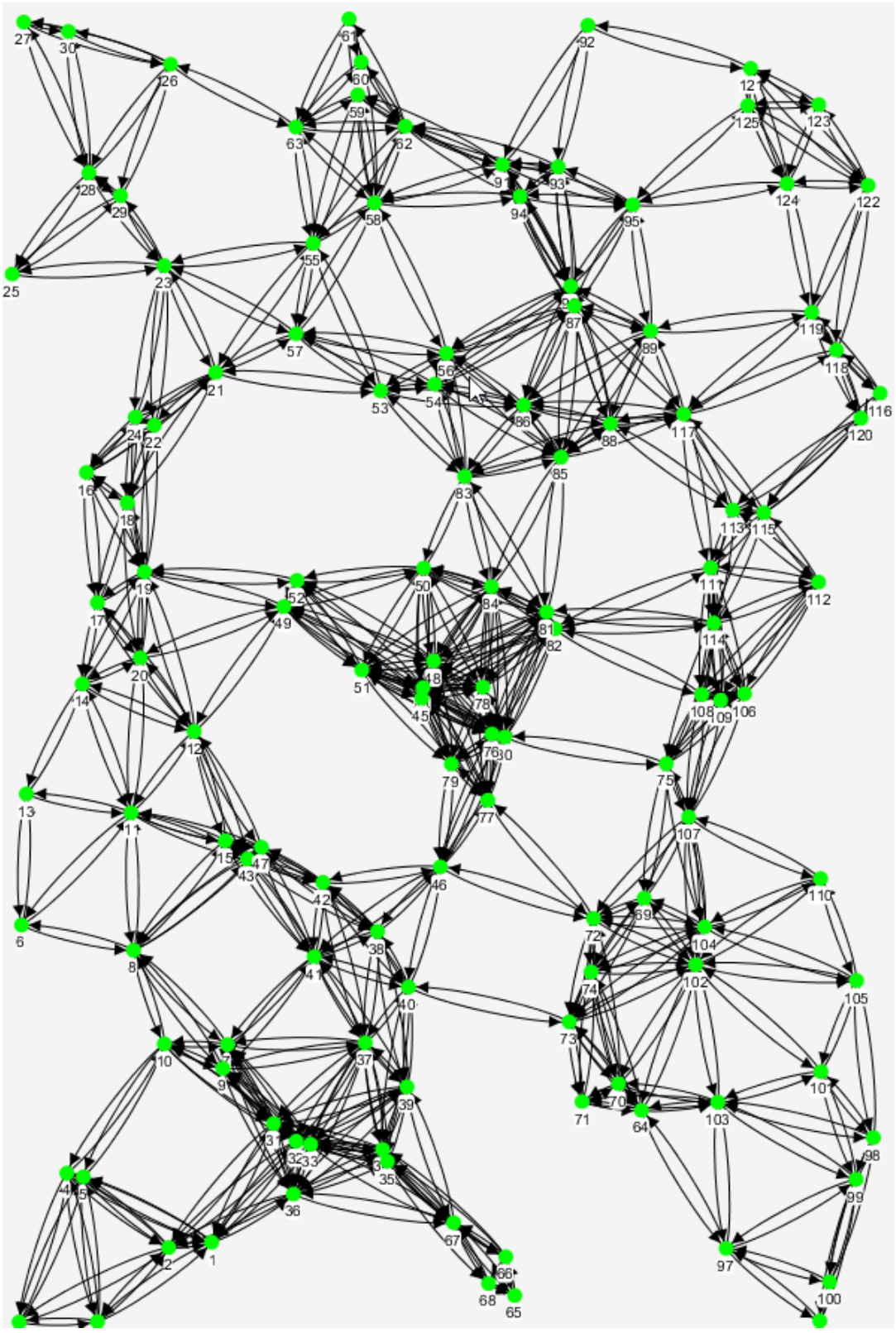


Figure 6.8: Snapshot from second LMAC result experiment. Note that this shows the state at 1185 ms. All nodes have successfully selected an available frame.

7 Case Study - AODVv2 Protocol

In this section we explore the Ad Hoc On-demand Distance Vector Version 2 (AODVv2) protocol [42], an ad hoc routing protocol. We will present an overview of this protocol, as well as a model describing it and use this model to visualize the behavior of the protocol.

Chapter Organization This chapter is organized as follows:

- In Section 7.1 we describe the AODVv2 protocol in broad terms,
- in Section 7.2 we provide an overview of the AODVv2 protocol model,
- in Section 7.3 we present the resulting visualizations and results gained.

7.1 Protocol Overview

In this section we will present an overview of the terms used in the AODVv2 protocol, as described by the internet-draft available on the document tracker for The Internet Engineering Task Force (IETF) by Perkins et al. [43]. Note that this protocol was formerly known as the DYMO protocol, but that this was renamed to AODVv2 in 2012, according to the internet-draft available from IETF [42]. Being an internet draft, this protocol is still under development, and thus the descriptions in this report is based on the most recent description at the time of writing. The purpose of this section of the report is to introduce the protocol in broad terms, such that we may use a protocol model based on this protocol later in the project. For the more complete specification of the protocol, we refer to [43].

The AODVv2 protocol is a routing protocol, which extends the traditional AODV protocol [44]. Thus, in order to describe the AODVv2 protocol, it is beneficial to understand the original AODV protocol. The AODV protocol is an Ad Hoc routing protocol, meaning that it is designed for routing in highly mobile, dynamic networks and mesh networks. Again, we will not give the complete specification for this protocol, but rather give the basic understanding of the terms and concepts of this protocol, such that we can reason about the visualizations we create later. For the complete specification of AODV, we refer to Perkins, Belding-Royer, and Das [44].

As the AODV protocol applies to mesh networks, nodes using the original AODV protocol initially have no knowledge of the topology of the network. In AODV a data payload is called a *Packet*. When a node wants to transmit a packet to a given destination, it must thus discover a route to it. This is done by flooding the network with what is known as a *Route Request*. This contains the ID of the original sender node as well as the ID of the destination node that the route request concerns. When the route request is propagated forwards, each node that acted as a relay for the route request, notes that it now knows a route to the original sender, if none was known before, and subsequently it continues the broadcast of the route request. When the route request reaches a node that knows a route toward the destination node, a *Route Reply* is transmitted back through the reverse route of the route request was sent. All nodes on this path populate their routing tables with the neighbor that they received the route reply from as the shortest path from them to the destination node.

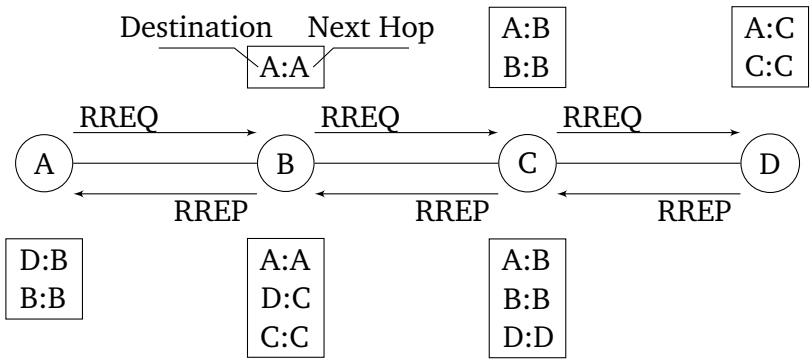


Figure 7.1: The concept behind AODV Route Request (RREQ) and Route Reply (RREP) messages. This figure is a copy of Figure 3 from Ahirwar, Verma, and Daksh [1].

From a single route request, all nodes affected by the route request know how to transmit to the originator through a fastest path at that time, and all nodes on the resulting route reply return path know how to transmit to the destination node as well. Note that in a dynamic network, the fastest path might change.

This concept is illustrated in Figure 7.1, where the tables above and below show the state of the routing tables of each node as the route request and route reply messages are propagated respectively. Pay special attention to the nodes B and C, which at the end hold routes not only to the nodes involved in the route request and route reply, but in fact to all nodes in the network.

There are also other types of messages, like *Route Error* which is used to flood the network when a link breaks, such that all routes depending on it can be invalidated. We will not elaborate further on these other types of messages.

According to Sommer and Dressler [46], the primary difference in the AODVv2 protocol over the AODV protocol is that it extends the route request and route reply messages, such that path accumulation is employed as they are being propagated. Instead of holding only the original node that transmits a message, a route request or route reply is decorated with all the nodes that it passes through on the way to the destination or original sender respectively. This means that more data is collected during the flooding process, which helps nodes along the path detect routes to all nodes that the message has passed through on the way. By allowing the nodes to obtain routes to all these nodes on reception of the message, the need for route request messages is reduced in the system, since more routes can be obtained in one go. Thus, a needed route may already be known when it would otherwise need to be requested. This comes at the cost of larger network packages, and potentially superfluous routes being added.

In the explanation for their model, Höfner and McIver [23] further elaborate that the AODVv2 protocol has some differences in the way sequence numbers are handled compared to the original AODV protocol. Sequence numbers are used to track how old a route request message is, where in the original all route requests was saved in a queue [23]. Furthermore, AODVv2 generates bidirectional routes, in the sense that all nodes on a route know how to reach all other nodes on the route as a result of the path accumulation and the way route replies are handled [23].

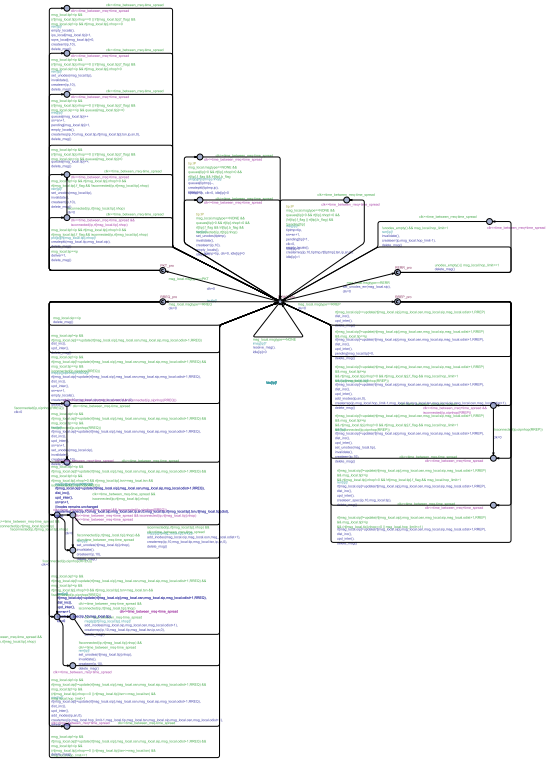
7.2 Model Overview & Compatibility

We will now present the AODVv2 (or DYMO) model developed by Höfner and McIver [23]. The goal of developing this model was to compare the performance of the AODVv2 protocol to the AODV protocol in a number of settings, and as thus the model is a complete representation of the protocol. There are three templates:

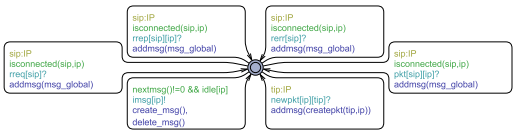
dymo contains the main node logic, and maintains the routing tables. This template is instantiated for each node.

queue handles a FIFO queue of size M, which stores messages that are to be sent. This template is instantiated for each node.

tester provides a way of configuring PKT transmission scenarios. This template is only instantiated once.



(a) Dymo template



(b) Queue template

(c) Tester template

Figure 7.2: The original AODVv2 model developed by Höfner and McIver [23]

To get a feeling of the size of this model, all the templates can be seen in Figure 7.2. Note that we do not expect this to be readable, but instead to give some sense of scale for the reader. The model is quite complicated, in that the main template, dymo, has 20 locations, and is rather verbose. However, one of the issues that we are trying to mitigate is that the user of VisuAAL may not know how to model in UPPAAL, and thus this model serves as an excellent

example for testing what kind of visualization can be gained without knowledge of UPPAAL modeling.

An obvious candidate for observable behavior in the protocol, is the number of messages that must be sent from each node. When a message is propagated around the network, it is added to a queue construct for each node, and thus, the flow of messages in the network can be visualized as the number of messages (Not only packets, but also route requests, route replies and route errors) waiting for transmission in each location is changed. This can be tracked as the global array `nodebuffersize`, which has as many elements as there are nodes plus an additional default node object.

7.2.1 AODVv2 Compatibility

The AODVv2 protocol model as presented by Höfner and McIver [23], is already an SMC compatible model. Thus, we only have to make it VisuAAL compatible.

This model contains an oddity that causes problems in VisuAAL; the node ids always start from 1 in this model, but all arrays are instantiated with a 0th element, which serves as a default "node". While there is no inherent problem with doing this, this approach becomes problematic when we try to read and write the topology. In this model, row and column 0 in the topology array are special in that the values are 0. There are no node 0 instantiations of the templates. This conflicts with the requirements we have set forth. We expect the topology to be a two-dimensional array with each dimension having the same number of elements as there are nodes in the network. While there are additional elements in all the arrays in this model as a result, the topology is especially important, since we generate this and parse it before the simulation is run, and are thus reliant on this being of the correct form. We need the instantiated array to be of this form, in order for the model to be compatible with VisuAAL.

Luckily, all access to the topology array is done through the function `is_connected(int i, int j)`, meaning that we are able to have an array of the correct form and emulate the first row and column, by looking at the input variables, since these entries always will be 0. This is shown in Appendix B.

There were a few minor time locks that occurred, due to inconsistencies in the guards in the system. We have attempted to fix these, but will not go through these fixes, as this quickly becomes much too detailed for the scope of this report. The resulting model is included in Appendix ZIP.

In addition, there are three other minor changes that we want to make:

- Remove the routing table parameter form the dymo template
- Change the tester template to transmit more messages
- Rename variables

The following sections will provide an overview of these changes.

7.2.2 Removing Parameter From dymo Template

The dymo template takes two parameters, a unique ip number, `ip`, and a reference to a routing table, `&rt`. The routing table references a global routing table of the form `rtentry art[N][N]`, where N is the number of nodes+1. Thus, each node has its own routing table which can

contain information about every other node in the network, but the routing tables are stored in global space. The templates are instantiated in the system declaration of the model such that each template is instantiated explicitly. The dymo template receives two parameters: a node id `ip` and a reference to `art[ip]`.

This is problematic, because the instantiations of processes does not automatically scale when `CONFIG_NR_NODES` is changed. The templates in the original model must rather be instantiated in the system declaration, making it incompatible with our tool, as described in Section 4. In order to change this, we have replaced all uses of this local reference parameter with a direct lookup in the global routing table, which is what the reference provided anyway. Thus, every usage of `rt` has been replaced with `art[ip]`, and the parameter has been removed. This means that the explicit instantiation in the system declaration can be removed and replaced by simply specifying the template name instead of the process names, which will instantiate a process for every node identifier. This is exemplified in Listing 7.1 and Listing 7.2, which shows the system declarations before and after the changes.

```
1 // Place template instantiations here.
2 a1=dymo(1,art[1]);
3 a1q=queue(1);
4 a2=dymo(2,art[2]);
5 a2q=queue(2);
6 a3=dymo(3,art[3]);
7 a3q=queue(3);
8
9 system tester,a1,a1q,a2,a2q,a3,a3q;
```

Listing 7.1: Original DYMO model system instantiation

```
1 // Place template instantiations here.
2 system tester,dymo,queue;
```

Listing 7.2: Modified DYMO model system instantiation

7.2.3 Changing tester template

Höfner and McIver [23] uses the model to explore a large number of topologies, and uses the tester template to set up and execute scenarios. In the model we downloaded, tester initially asks node 1 to transmit a packet to node 2, and after a period of time to transmit a packet from node 1 to node 3.

Having a template that dictates when the environment asks the nodes to transmit packets is a good idea, since this is separate from the protocol behavior itself, but the simple behavior configured in the original model makes for boring visualization. To make for more interesting visualization, we have changed the tester template such that it continuously injects packets in the network, by periodically randomly selecting a source and destination node, and inject a packet to the source node, to transmit. The period between each injection, as well as the transmission speed of every other aspect of the model, is governed by the variables `time_between_rreq` and `time_spread`.

This change to the tester substantially changes how the model operates, and makes it difficult to repeat simulations, but it also results in lively communication on the network and means that the visualization looks more like a realistic scenario, at least in our opinion. If a specific scenario is needed, the model may be reconfigured by changing this template.

7.2.4 Renaming Variables in the AODVv2 Model

VisuAAL needs to be able to read the topology, the number of nodes and a number of other variables. The renamings we have performed are summarized in Table 7.1. The variables do not need to be moved to different scopes in order for the model to be VisuAAL compatible.

	Original Variable Name	New Variable Name
Topology	`topology[][]`	`CONFIG_connected[][]`
Number of Nodes	`N`	`CONFIG_NR_NODES`

Configuration Variables	`M`
	`time_between_rreq`
	`time_spread`
		`CONFIG_MAX_SEQ_NR`
		`CONFIG_time_between_rreq`
		`CONFIG_time_spread`

Output Variables	`nodebuffersize[]`	`OUTPUT_nodebuffersize[]`

Table 7.1: Renamings necessary in the AODVv2 protocol model

7.3 Results for AODVv2 Protocol

In this section we will present an example visualization for the AODVv2 protocol. In this case study, we are interested in the traffic in a network using the AODVv2 protocol, and as such we will try to visualize the number of messages waiting for transmission on each node over time.

We generate the topology shown in Figure 7.4, where a few nodes must mediate all transmissions between the top and bottom subgraphs. The top and bottom subgraphs have 23 and 21 nodes respectively. The precise configuration resulting in this topology is not important, what is important is that this topology should result in a lot of traffic across the bridge nodes. In a topology where a few nodes act as bridges between highly connected clusters of nodes, we expect that the bridge will have a much larger backlog of transmissions than other nodes. The goal of this is to visualize to what degree this happens in a topology, and also to see how this changes over time, as the nodes discover routes. When routes are created, there are less need for route requests and route replies, meaning that less transmissions in total will be needed to transmit a given packet. Recall that the tester template is set up to inject a new packet at a random time between 35 and 55 mtu, with a uniform distribution.

Because of the way the tester template is set up, the destination and transmitter nodes should be selected from a uniform distribution over node ids. Thus, packets sent from one subgraph will either need to be sent to a node within their own subgraph, in the bridge or in the other subgraph. The transmissions to the other subgraph have to go through the bridge. There should be approximately the same probability of a node from the top selecting a destination in the top as a destination in the bottom, and vice versa for the bottom sending to the top. There are 48 nodes in the topology.

As explained in Section 7.2, we will track the messages waiting to be sent through the output variable `OUTPUT_nodebuffersize`, which results in an integer value of the size for each node's queue.

Experiment 7.1 - AODVv2 Experiment

In this visualization, a gradient from a light color to a strong color is appropriate, since we want to highlight large differences between the number of messages in the queue. We construct the following VQ expression to distinguish this:

```
[lightgray:0, red:25] OUTPUT_nodebuffersize
```

This will mark nodes with 0 messages waiting with a light gray color, and nodes with 25 messages accumulated red. 25 just so happens to be the maximum queue length in the simulation we ran, and thus we have chosen it as the upper bound for the gradient. Figure 7.4 shows a screenshot at time stamp 1000 mtu, where the bottom of the bridge is clearly illuminated in red, as a result of many messages waiting for transmission here. The simulation is bounded at 1000 mtu, where a packet is injected every 35 to 55 mtu, meaning that there are between 18 and 28 packets being sent in the simulation. It took VisuAAL roughly 3 minutes to run the simulation with 48 nodes in the topology. The entire simulation can be found in Figure 7.3, and the simulation file can be found in Appendix ZIP, where it is called `AODVv2_result_1.sim`.

Figure 7.3: Visualization of Experiment 7.1. The more red a node appears the more messages are in the message queue.
The QR Code will redirect to the following link:
`https://youtu.be/D8vq5gCmHi4`

Intuitively we expected that the bridge would almost constantly be congested. Since we have set the model up so that messages are sent to random destination, we expected that many messages would accumulate at the ends of the bridge, because these nodes have many incoming edges from the subgraph they belong to, but only a single edge toward the other subgraph. When we look at the simulation, an interesting trend emerges. There does indeed seem to be congestion on the ends of the bridge, but there is actually congestion within the subgraph as well. This surprises us, as there is no mechanism as far as we are aware for nodes refusing a message for the queue. Since the transmitter and destination nodes are randomly selected, we would not expect any congestion as such within the subgraph. We believe the reason behind the congestion in the subgraphs to be linked to the flooding that happens during route request messages, combined with the complex interactions that result from the routing tables being updated. Recall that as soon as a node knows the wanted destination for a route request it will respond with a route reply without the route request needing to reach the destination. Further recall that the routing tables are updated when nodes act as relays for route request and route reply messages, and that routes to nodes are thus rather quickly propagated.

To see if this was simply a fluke of of the simple simulation, or whether there seems to a general trend of congestion with multiple simulations, we ran the same simulation but for 100 different runs, so that we could examine the average simulation. This took approximately 1 hour and 10 minutes to execute on our Linux server. We use the same topology as before, but the following VQ:

```
[green:0, red:11] OUTPUT_nodebuffersize
```

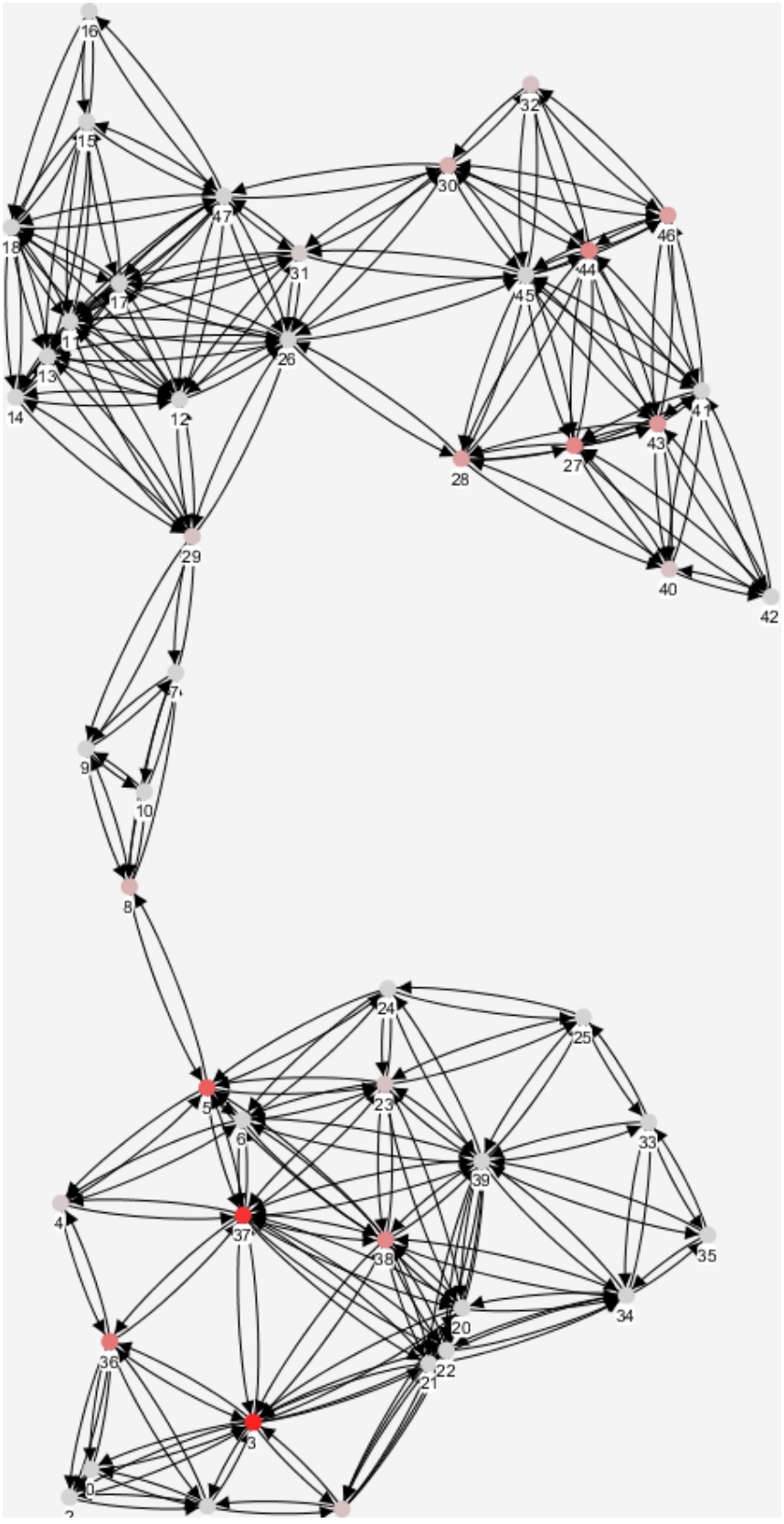


Figure 7.4: Screenshot from simulation for AODVv2 protocol model at 1000 mtu

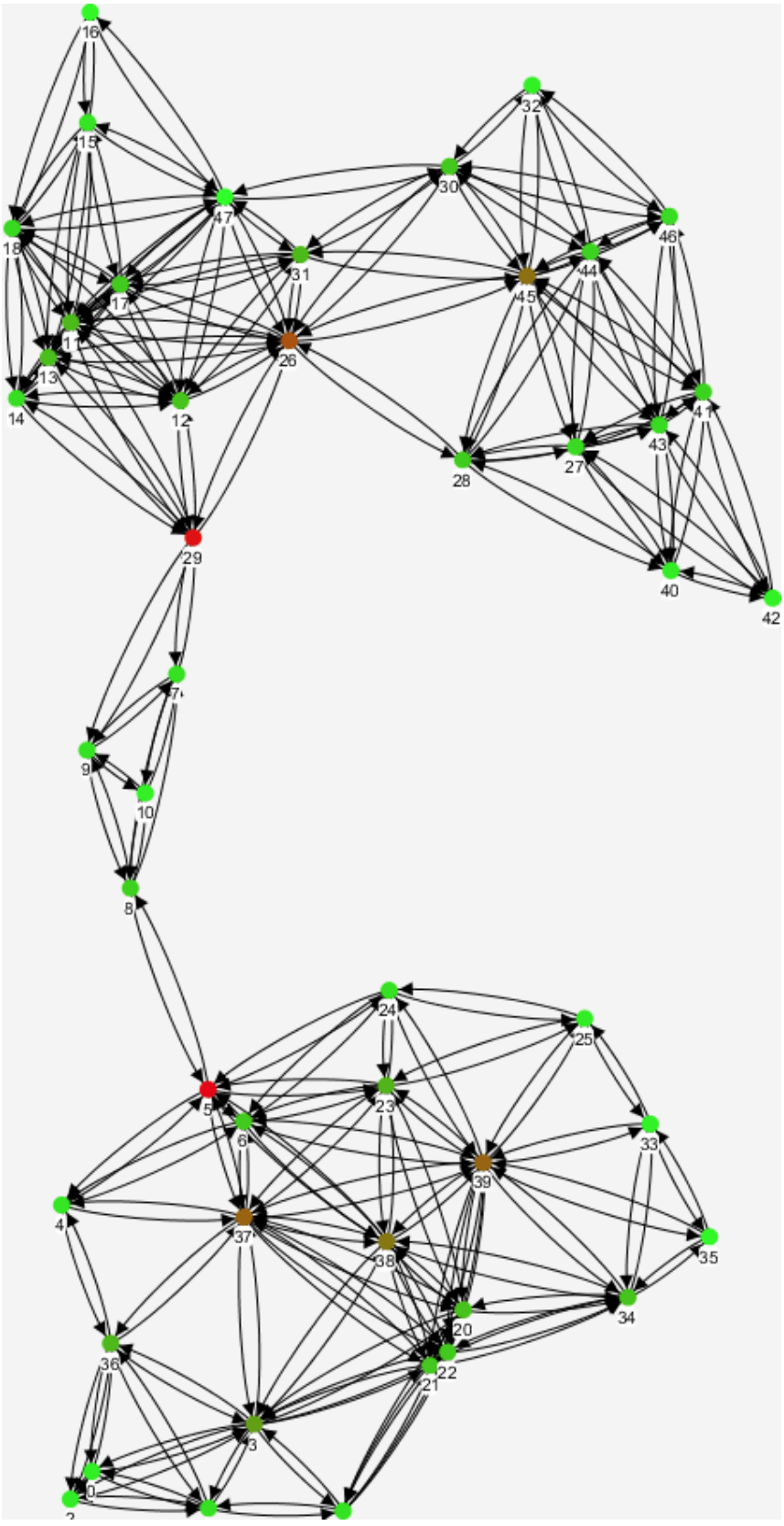


Figure 7.5: Screenshot from average simulation of 100 simulations for AODVv2 protocol model at 1000 mtu

Note that we in this VQ have changed the colors to be a gradient from a green to red, to emphasize the changes. We have also decreased the max value in the VQ since the highest average value was around 11.

A screenshot from this simulation can be seen in Figure 7.5, and the video of the simulation can be seen in Figure 7.6. The simulation file is called `AODVv2_result_2.sim`. In the average of the 100 simulations, we can see that the nodes in each end of the bridge have more messages in the message queue than the average node in the network. Furthermore the nodes located towards the center of the subgraphs always have a tendency to have many messages in the message queue.

Figure 7.6: Visualization of Experiment 7.1 repeated with 100 simulations, showing the average simulation. The more red a node appears the more messages are in the message queue.
The QR Code will redirect to the following link:
`https://youtu.be/ZLJgu-2h0yE`

We contacted the authors of the model, and presented an earlier version of VisuAAL showing a similar simulation for a similar topology to them. Rob van Glabbeek replied that he was surprised to see congestion happening in the areas where it did, just as we were. Specifically, he wrote the following:

> I find your visualisation very interesting, as it shows congestion in a realistic situation. One of the perhaps surprising outcomes I see is that congestion happens all over the place; it is less centered in the narrow corridor than one might have expected.
>
> *Rob van Glabbeek [17]*

By using visualization we could quickly get an overview of a quite complex situation. We realized that the behavior of the protocol model is different from what we and the author of the protocol model expected.

8 Case Study - Neocortec MAC Protocol

In this section we look at a Medium Access (MAC) protocol model that we created in [29] based on our understanding of Neocortec's MAC Protocol [27]. We want to stress that the model and our understanding of the protocol is not the same as Neocortec's protocol, since we do not have all the details. We based our understanding of the protocol on the patent for the MAC protocol owned by Neocortec [27], as well as conversations with Neocortec, but there are substantial discrepancies not only in which features we have included, but also in the configurations we run. This means that the results we obtain can not be compared to Neocortec's implementation. When we refer to the MAC protocol in this chapter, we are talking exclusively about our understanding of it.

We will not explain the details of the MAC protocol, since it includes some confidential information. We will however present an overview which should be sufficient to read and understand the experiments that we present. For more detail we refer to [29], where we explain the protocol and the model that we created in greater detail.

We will not present the model for the MAC protocol in this report, since the confidential details are implemented in the model, but it is available in Appendix ZIP. To give a sense of the model size, we remark that the model has 7 templates. 6 of the templates are instantiated for every node in the system. The templates each have between 5 and 10 locations.

The model of the MAC protocol is SMC compatible and for this reason it is only required to rename configuration and output variables. We will not further discuss changes to the model.

Chapter Organization This chapter is organized as follows:

- In Section 8.1 we briefly describe the terms used in the protocol,
- in Section 8.2 we present some experiments and the results obtained from applying VisuAAL to the model.

8.1 Protocol Overview

The MAC protocol used by Neocortec serves the purpose of broadcasting data to neighbor nodes. The key thing is that it happens in intervals, such that the nodes can sleep as much as possible in order to save energy.

Figure 8.1 shows an abstract overview of the MAC protocol. Initially a node performs an initial full beacon scan, and then periodically performs full beacon scans, beacon scans, sends data and receives data. Whenever a node is not performing one of these tasks it goes into a low-power sleep mode. When doing beacon scans the node listens for other nodes that it can schedule data transmissions for and can be synchronized to. The beacon scan is divided into beacon slots where the node either receives or sends scheduling information. Since the nodes sleep in between tasks, they need to be synchronized, to know when to wake up for the next

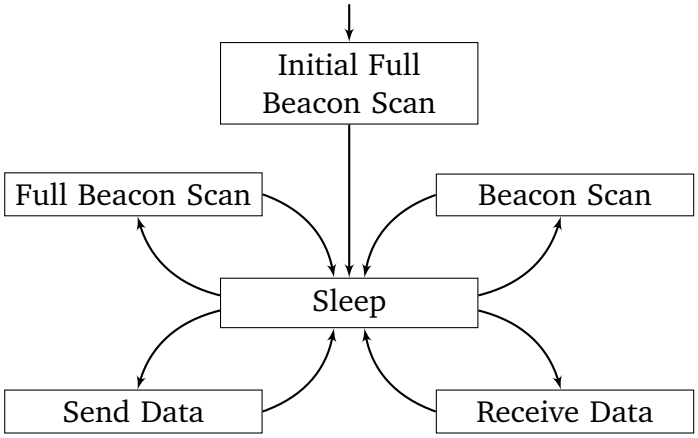


Figure 8.1: Abstract flowchart over Neocortec MAC protocol. Original from [29].

task. The node can only receive user data from neighbors if the node has received schedule information of when the neighbors are going to send this data.

Beacons

To allow a network of nodes to function without a central node, the MAC protocol takes advantage of synchronizing nodes to other nodes' beacons. Beacons are short synchronization messages for nodes within range. A beacon is sent from a node to inform other nodes when it will be transmitting data packages. The nodes receiving a beacon then schedule when to listen for data transmissions from this node.

Beacon Scan

For the beacon method to work, each node needs to know when beacons are transmitted. This is where the synchronization of nodes becomes important. If the nodes are synchronized in regards to the start time of a beacon period, it is possible to have a beacon scan wherein the nodes transmit and receive beacons, as shown in Figure 8.2. This transmission and reception happens in a number of slots, and the beacon scan is significantly shorter than that of a beacon period. By knowing when the transmission should start, and how many nodes are going to transmit, it is possible for the nodes to turn off their transceivers outside of the scheduled beacon transmission times. Periodically the beacon period is replaced by a full beacon scan, to discover new neighbors.

Beacon Slots

Each beacon scan is split into a number of beacon slots. When a new beacon scan begins, each node chooses a random beacon slot from a preconfigured set of possible slots. Each node then transmit their own beacon in their selected slot, and receive beacons of other nodes in the remaining slots. This method of choosing slots is similar to the Slotted ALOHA protocol [28].

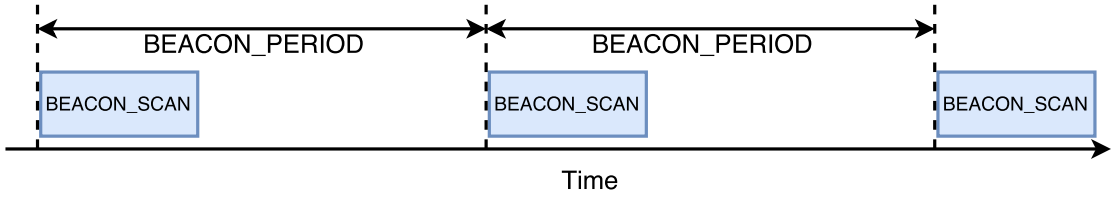


Figure 8.2: Beacon scans for a node and how they relate to beacon periods. BEACON_SCAN denotes the length of the beacon scan, and BEACON_PERIOD denotes the time between two beacon scans. Originally published in [29].

Full Beacon Scan

In order for nodes to discover networks that they are not connected to, the protocol utilizes special beacon periods called full beacon scans, which work like normal beacon periods, except that the node keeps listening for beacons outside the beacon scan. The node listens for beacon transmissions that start at other times than the nodes own beacon schedule. The node can schedule data transmissions from nodes of other networks and potentially synchronize to another network. Note that a beacon scan is still performed during a full beacon scan, and that the length of a full beacon scan is exactly equivalent to that of a beacon period.

Initial Full Beacon Scan

When a node is first enabled, the node does not have any knowledge of its neighbors, and to obtain this and to start networks quickly, the protocol does a initial full beacon scan. This is a special beacon scan, where a node tries to discover a network to connect to. In order to do so, it performs a number of full beacon scans interleaved with normal beacon periods, to faster discover neighbors that are out of sync.

Scheduled Data Transmissions

The data schedule of a node is independent of the beacon schedule for the protocol. A node has a unique preconfigured data interval between data transmissions. This interval is retained throughout the life of the node, such that the period between each scheduled data transmission from a node is constant, but different from other nodes. The purpose of having different data intervals for each node is to avoid nodes repeatedly colliding.

8.2 Results - MAC Protocol

In this section we will present the results obtained from visualizing the Neocortec MAC model. There are two main aspects we want to visualize, one for nodes, and the other for the edges between nodes.

Both of these aspects will be visualized on a randomly generated topology of size 61 nodes, which can be seen in Figure 8.4.

For nodes, we want to visualize which slot they have taken, such that it is possible to see how prevalent the problem of nodes selecting the same beacon slot is. If nodes choose the

same beacon slot, they will have a collision in their next beacon period, and thus any nodes listening to both will receive neither transmission.

This is done by tracking the output variable `OUTPUT_chosen_slot`, which is an integer between 0 and the largest allowed slot. We use a configuration with 8 beacon slots.

For edges, we want to observe when nodes discover each other. We want to see whether a node is aware of each of its neighbors, which gives an overview of the available communication channels for the nodes.

The array `OUTPUT_data_is_scheduled` tracks whether a node is aware of a neighbor's next data transmission, in the sense that `OUTPUT_data_is_scheduled[x][y] > 0` is true if `x` is aware of the data transmission coming from `y`, and thus will `x` wake up to receive the data transmission from `y`. By tracking this, we get an overview of which nodes will be able transmit to which nodes, and thus of the connectivity of the network.

Technically, we are able to show both edge visualization and node visualizations at the same time, but to improve readability in this report, we have chosen to split it over two experiments.

As a third experiment, we want to highlight some of the features of the MAC protocol, which help with power preservation and how we can visualize this. The experiment visualize the different operating modes on each node, to get a better overview of how the nodes interact with each other.

Experiment 8.1 - MAC Protocol - Node Visualization

To visualize the beacon slots in each node, we will use the following VQ, such that nodes will be colored uniquely according to the slot they have chosen. If nodes do not have a chosen slot, they will be colored black, the default color for nodes. This is for instance true during the initial beacon scan.

```
[red:0, orange:1, yellow:2, green:3, blue:4, purple:5,
        wheat:6, gray:7, black:*] OUTPUT_chosen_slot
```

We run the visualization up to 60000 mtu, which takes 15 minutes to run in VisuAAL. This can be seen in real time in Figure 8.3, or on the included simulation file in Appendix ZIP named `MAC_result.sim`.

Figure 8.3: Visualization of Experiment 8.1, which visualizes what slot each node has selected. If the node is colored black, it will not transmit a beacon.
The QR Code will redirect to the following link:
`https://youtu.be/fc6pMlLs0ro`

Figure 8.4 shows a snapshot from time stamp 7074.8 ms. An enlarged version of part of the top right part of this figure is presented in Figure 8.5, to illustrate how nodes 55 and 59 have chosen the same beacon slot, 5, and are thus both marked purple. This will lead to a collision on the next beacon scan. By using visualization, a quick way to understand how the nodes select their slots is obtained, and it is visible how different configurations change the probability of adjacent nodes selecting the same slot.

In addition, we have observed that getting the nodes synchronized can be difficult, such that their beacon periods start at the same time. The timings between the nodes can be seen graphically in the visualization, as nodes select a slot at the beginning of every beacon period, and thus the changing colors in each node marks the beginning of a beacon period. When the simulation is played back, more nodes become synchronized over time, and thus blink synchronously.

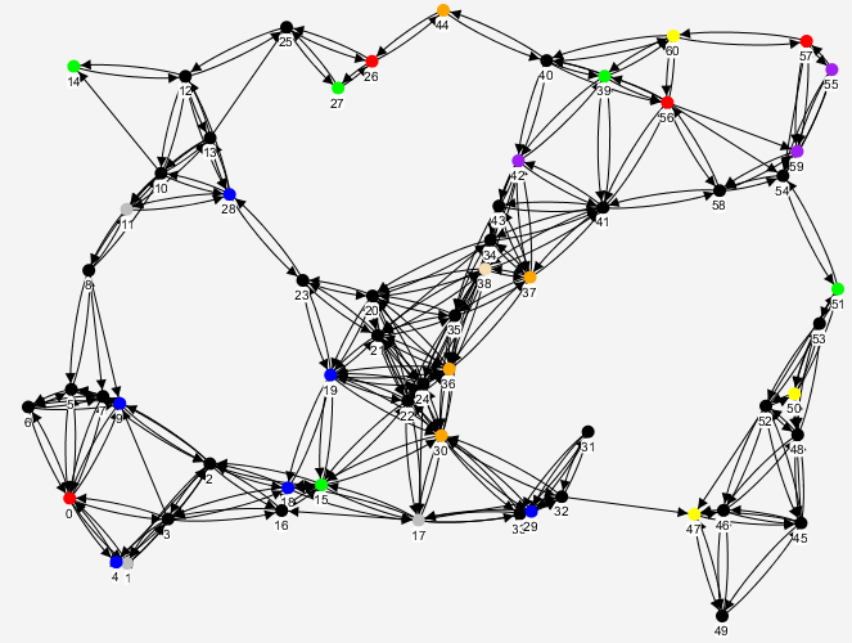


Figure 8.4: Snapshot from Experiment 8.1. This shows the state of which nodes have selected what slot at time stamp 7074.8 ms.

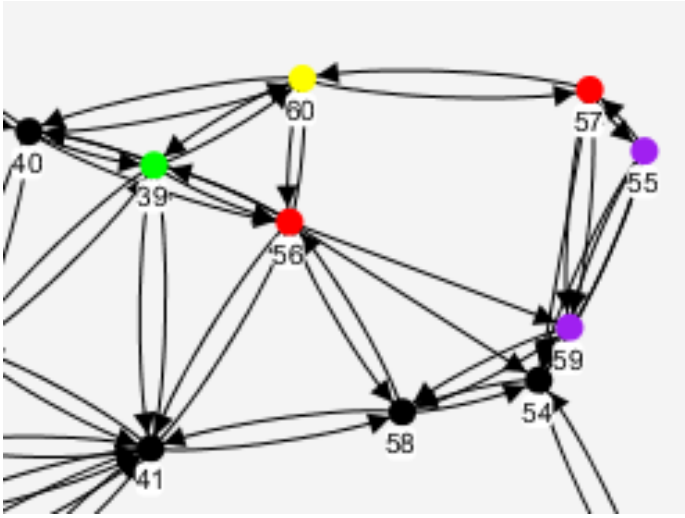


Figure 8.5: Enlarged and cut version of Figure 8.4

Experiment 8.2 - MAC Protocol - Edge Visualization

In this experiment we will show how the nodes become aware of their neighbors' scheduled data transmissions. This can be used to see the connectivity of the network, since only by scheduling data transmissions may the nodes ever send data packages through the network. The following simple VQ uses the fact that the gradient is optional to color edges red when the source node is aware of the coming scheduled data transmission from the destination node, and black if not:

```
OUTPUT_data_is_scheduled
```

We use the same simulation as Experiment 8.1, and present a snapshot with this new VQ in Figure 8.7, which shows which nodes are prepared to receive data at time stamp 17084.5 ms. The full visualization can be seen on Figure 8.6 where it is replayed in real time.

On Figure 8.7 we can see that we have two disconnected networks, which cannot communicate. Recall that the red edges visualizes edges where data can be transferred. This clearly shows the importance of the full beacon scans in order to discover unsynchronized networks and enable data transmission across the whole network.

Figure 8.6: Full visualization of Experiment 8.2. It shows which of the nodes' neighbors that have been scheduled to receive data from.
The QR Code will redirect to the following link:
`https://youtu.be/Y5DMfAp8f3A`

In addition to this, we visualized both VQ expressions simultaneously from Experiment 8.1 and Experiment 8.2 as shown in Figure 8.8.

Figure 8.8: Visualization of Experiment 8.1 and Experiment 8.2 in the same video.
The QR Code will redirect to the following link:
`https://youtu.be/_lQshtxUayU`

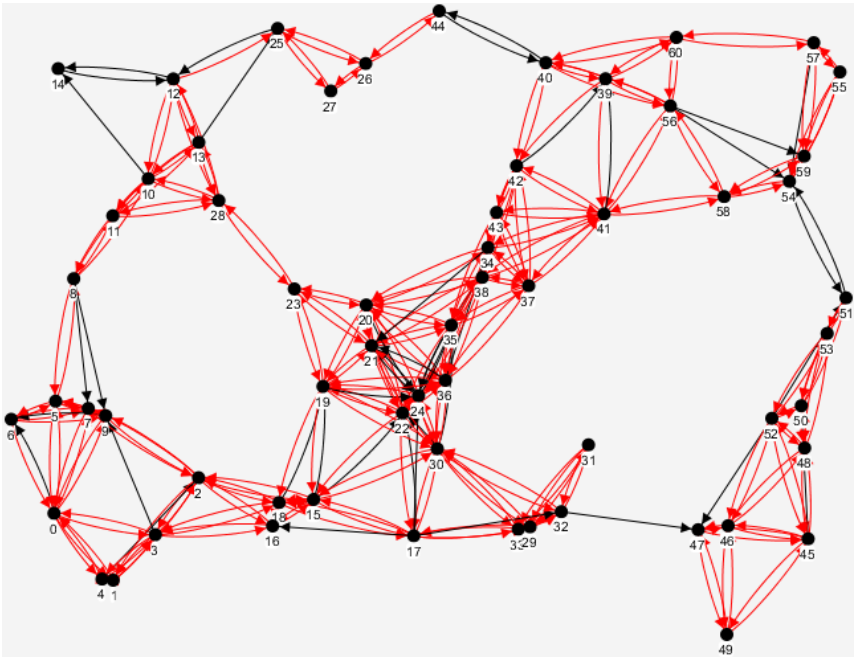


Figure 8.7: Snapshot from Experiment 8.2. This shows the state of which nodes have scheduled their neighbors data transmission at time stamp 17084.5 ms.

Experiment 8.3 - MAC Protocol - State of Nodes

In this section we look at how we can visualize the state of each node. Ivanov [25] have annotated the protocol model with power consumption which we can use to distinguish the state of each node. The model is configured to have a 20 mA current draw when transmitting and 19 mA when receiving. In reality, the current draw for these two states should be approximately the same, but in order to illustrate the current draw in the visualization and explain the concept, we have introduced this difference in power draw. Additionally the current draw when sleeping is 0.5 µA. We are not claiming that this is a perfect annotation to show precise energy consumption, but these values are based on data from the CC1110 transceiver datasheet [47]. We have come to understand that Neocortec use a transceiver very similar to this. Note that there are many factors to consider for the current draw for different operating modes and thus realistically there is not only one value for e.g. transmitting.

In this experiment we want to highlight some of the features of the protocol, where one of the main purposes is to save energy. We use the following VQ expression for the variable `OUTPUT_battery_i` which describes the current draw, where nodes turn red when transmitting, blue when receiving and green when sleeping:

```
[red:20, blue:19, green:*] OUTPUT_battery_i
```

For the experiment we generate a topology with 66 nodes. We simulate the protocol model for 50000 ms, which takes around 30 minutes of query execution time. We show a cut snapshot visualization of the simulation in Figure 8.9. Note that this is only part of the topology, as the entirety of the topology is not important yet. The visualization is shown at time 33475 ms,

where we can see one of the interesting features about the MAC protocol model. Nodes are only listening when they know a neighbor is going to transmit data. We can see this because the nodes far from the red transmitting node are green, thus still asleep, while the closer blue nodes are listening. The blue nodes are all neighbors with the red transmitting node. The full simulation is shown in Figure 8.10. The simulation can be also loaded into VisuAAL and shown, it is found in Appendix ZIP named `Energy_state.sim`.

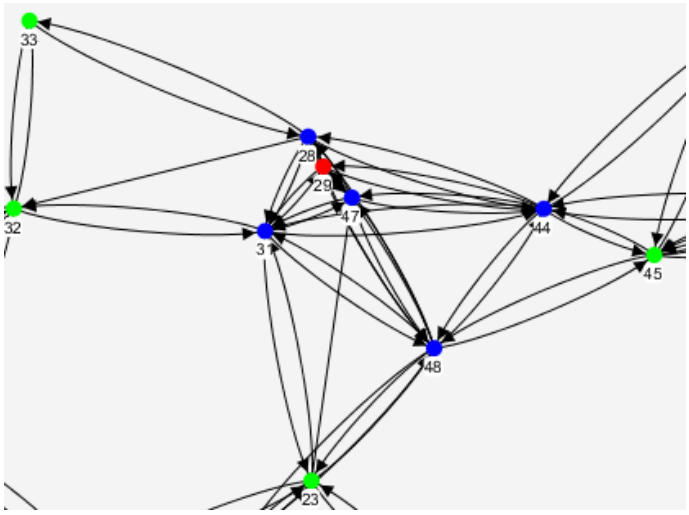


Figure 8.9: Snapshot at time 33475 ms. The red node is transmitting, blue nodes are receiving and green nodes are sleeping.

Figure 8.10: Energy usage of nodes is visualized for the Neocortec MAC protocol model in 10 % of real time. Nodes send for only a period of 2 ms, making it hard to see. The QR Code will redirect to the following link: `https://youtu.be/hPTHiciuhqQ`

As expected the nodes sleep most of the time, and once in a while they wake up to either send and receive beacons or data. Interestingly, we are able to use the current draw data to visualize the behavior of the nodes in terms of sleep, transmission and reception, and in this way understand how the nodes interact. The simulation shows how the nodes only listen to those of their neighbors, that have scheduled data transmissions via beacon scans.

If we consider the current draw as an indicator of the state of the nodes, a few interesting things about the protocol behavior can be gleamed from looking at this simulation. One thing that can be seen in the simulation is how initially nodes all perform the initial full beacon scan, where all nodes turn blue with some offset and after three periods begin to transmit. This happens during the first few seconds of the simulation. We will not present a snapshot of this observation, but we invite the reader to either watch the video in Figure 8.10 or load the simulation.

At time 45212.1 ms, a beacon scan happens as shown in Figure 8.11, where around half the nodes participate. For this snapshot we added the following VQ expression for edges, to show which nodes that have scheduled their neighbors:

```
[lightgray, black] OUTPUT_data_is_scheduled
```

The nodes doing a beacon scan are particularly interesting, considering that this can almost only happen when all participating nodes are synchronized, and following the same schedule. We can see that there are 4 disconnected networks in the snapshot, where the entire top right network is synchronized. Here we can again see the need for full beacon scans, that will allow nodes to discover other networks.

In Section 10.2 we go more in depth with the energy consumption of the nodes and how we can visualize that.

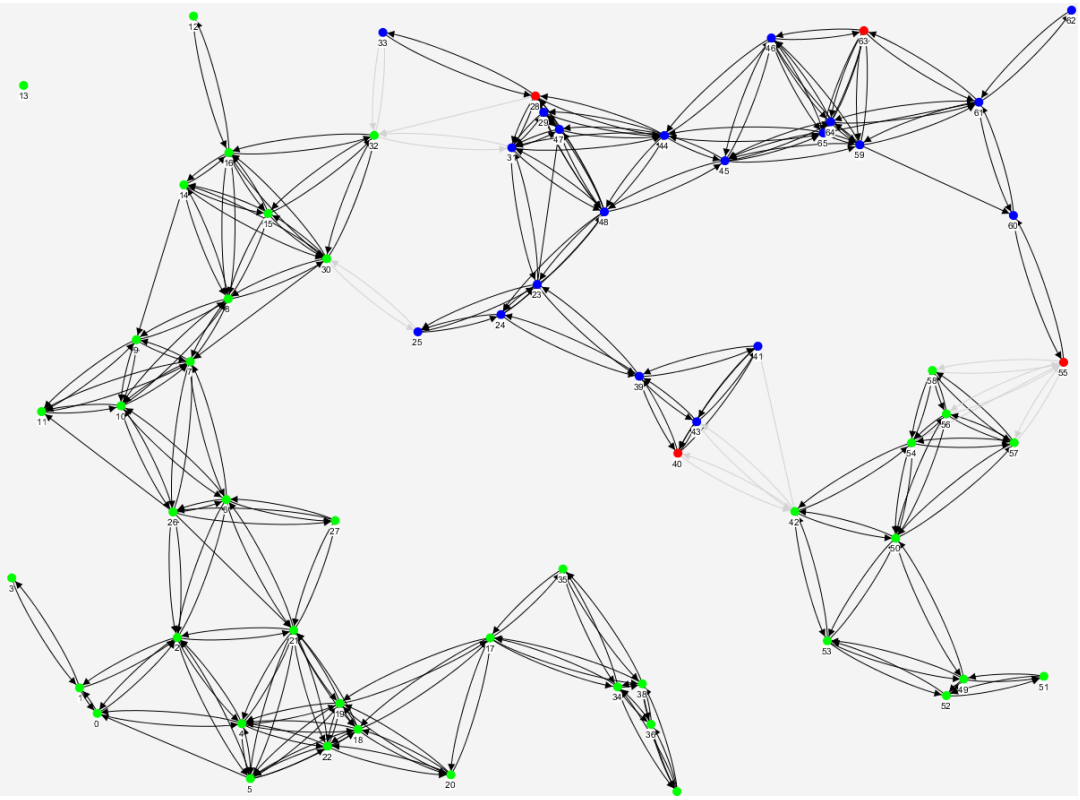


Figure 8.11: Snapshot simulation at time 45212.1 ms. Red nodes are transmitting, blue nodes are receiving and green nodes are sleeping. Black edges indicate where neighbor nodes are aware of each other.

9 Case Study - Neocortec Routing Protocol

In this chapter we look at the Routing protocol model that we created in [29], which is based on our understanding of Neocortec's Routing protocol [26]. We want to stress that the protocol model and our understanding is not equal to Neocortec's protocol, since we do not have all the details. We based our model and understanding of the protocol on the patent for the Routing protocol owned by Neocortec [26], as well as conversations with Neocortec, but there are substantial discrepancies not only in which features we have included, but also in the configurations we run, which are not the same as Neocortec use.

This means that the results we obtain can not directly be compared to Neocortec's protocol. When we refer to the Routing protocol in this chapter, we are talking exclusively about our interpretation of Neocortec's Routing protocol.

We will not present the model nor explain every detail of the Routing protocol, since it contains confidential details. We will however present an overview which should be sufficient to read and understand the experiments that we present. For more detail we refer to [29], where we explain the Routing protocol and our model of it in great detail.

To give a sense of scale, we remark that the model has two templates, where one, the SetupTemplate, is instantiated once, and the other, Node is instantiated for every node. The SetupTemplate has two locations, and is used to instantiate arrays. It performs this instantiation at time 0 and is never active again. The Node template has 8 locations and 18 edges. The model of the Routing protocol is SMC compatible and for this reason it is only required to rename the output and configurable variables which are desired for visualization. The changes to the model are trivial and we will not discuss these changes further.

Chapter Organization This chapter is organized as follows:

- In Section 9.1 we briefly describe the Routing protocol,
- in Section 9.2 we present the results obtained from applying VisuAAL to the model.

9.1 Protocol Overview

The goal of the Routing protocol is to route data from one node to another. For this protocol there are normal nodes and sink nodes. When data is routed, only sink nodes can be a final destination. All the nodes have only the information of which neighbors they have, and which intermediate node to send to in order to reach each sink node in the network. The routing information for nodes is contained in a *routing table*, and is updated regularly.

An important detail is that the nodes' data transmission schedules are not synchronized. The nodes all randomly select offset for transmission, meaning that nodes will likely have a different transmission delay compared to their neighbors', such that collisions between them eventually are resolved.

9.1.1 Race Numbers

The protocol uses *race numbers*, which are broadcast from sink nodes at regular intervals and flooded through the network. Receivers of a race number store it if the number is the newest race number from the sending sink node. Since nodes store the race number only if it is newer than the currently stored race number, it will most likely be a fastest route to the sink node. For instance if a node receives a race number from a neighbor and later receives the same race number from another neighbor, it knows that the first received sender is likely a fastest route toward the sink node. Note that this may change due to nodes sending at different intervals. The race number is incremented for every race number the sink node sends through the network, with the added caveat that when a maximum race number is reached, the race number is reset, such that a cyclical scheme is used to transmit race numbers. Note that only sink nodes increment race numbers, and the ordinary nodes simply retransmit their race number at regular intervals.

9.1.2 Routing Table Construction

Each node maintains a routing table with race numbers with an entry for each sink node in the network. Each entry of this routing table contains the id of a sink node, the newest race number received for this sink node and the neighbor that sent this race number. When a node needs to transmit a data package to sink node s_1, it transmits directly to the node specified by the routing table at s_1, and in this way propagates the data toward the sink node. Note that this means that the protocol (and model) assume bidirectional communication.

9.2 Results - Routing Protocol

We will now visualize the behavior of the Routing protocol model which we presented in [29]. For these experiments, we want to again stress that the configuration variables we use are in no way representative of the parameters used by Neocortec, and have been chosen by us for our model. Thus, the results we obtain give information about our understanding of the protocol and our model only. The first thing we will look into is the progression of race numbers and the second is the routing of a package to a sink node.

Experiment 9.1 - Routing Protocol - Race Number Visualization

To track race number progression in simulations we add some output variables to the model to support this. For this experiment, we setup the model with 2 sink nodes that will flood the network with race numbers. The largest allowed race number is 5 in this configuration. The variable `Node.OUTPUT_first_sink_race` and `Node.OUTPUT_second_sink_race` contains, for every node, the race numbers of the first and second sink nodes which have id 0 and 1 respectively.

We use the following VQ expression to visualize when nodes have received a race number from the sink nodes. The expression uses 3 conditional operators in order to differ between the combinations of data. The expression returns 3, which is the color green, if a node has received a race number from both sink nodes. The expression returns 2, which is the color red, if a node has only received a race number from the first sink node. The expression returns 1, which is the

color blue, if a node has only received a race number from the second sink node. In the other case where nodes have not received any race numbers, the value is 0 with the color lightgray.

```
[blue:1,red:2,green:3,lightgray:*]
Node.OUTPUT_first_sink_race > -1 && Node.OUTPUT_second_sink_race > -1 ? 3
: Node.OUTPUT_first_sink_race > -1 ? 2
: Node.OUTPUT_second_sink_race > -1 ? 1
: 0
```

In Figure 9.2 and Figure 9.3 we show two snapshots of a simulation of the routing protocol after 3 and 5 seconds respectively, with the VQ described above. The topology contains 63 nodes in total and it takes less than a minute to generate the simulation. The full visualization video can be found in Figure 9.1, and in Appendix ZIP where we have also included the simulation file, which is named `routing_exp_1_a_unmodified.sim`.

Figure 9.1: Visualization of Experiment 9.1 which visualizes when nodes receive race numbers from sink nodes. Please note that the video is shown in 50 % real time, and the first 8 seconds are shown.
The QR Code will redirect to the following link:
`https://youtu.be/H3vAnQpw3vE`

The visualization of the model shows some interesting aspects of the protocol model. Nodes send all race numbers at some specified interval. We can see when simulating the model, that the interleavings of these intervals affect how fast the race number progress in the network. From the visualization we can see there is a sequence of nodes where they all send right after the node before it, such that a race number can pass several nodes in a very short time frame, compared to when nodes wait an entire interval before sending the race numbers.

In the model the nodes do not initialize until they receive the first race number. After the initialization, the nodes send a race number after waiting exactly its sending interval. Because of this, the propagation of race numbers is slow to begin with, since that for every jump of a race number, the entire interval of each node will pass. When nodes are already initialized the interleavings of intervals can cause race numbers to propagate faster. In a scenario where nodes cannot lose packages, the slowest propagation of race numbers happens when for every jump, the following node waits its entire interval before sending it.

We believe that race numbers will initially propagate faster in the network, if they from the start follow the intervals with the random interleavings between nodes. We made a small change to the model to test this, where nodes follows their intervals from the beginning. When a node receives the first race number, it will instead of waiting an entire interval, only wait the remaining time for the current interval.

We continue the experiment with the topology shown in Figure 9.2. The simulation of the revised model, which is included in Appendix ZIP named `routing_exp_1_b_modified.sim`, shows that the race numbers progress faster to begin with. From our visualization we can see for the original model, it takes around 6 seconds to propagate race numbers from both sink nodes to all nodes. For the revised model it takes around 5 seconds to propagate the race numbers to all 63 nodes.

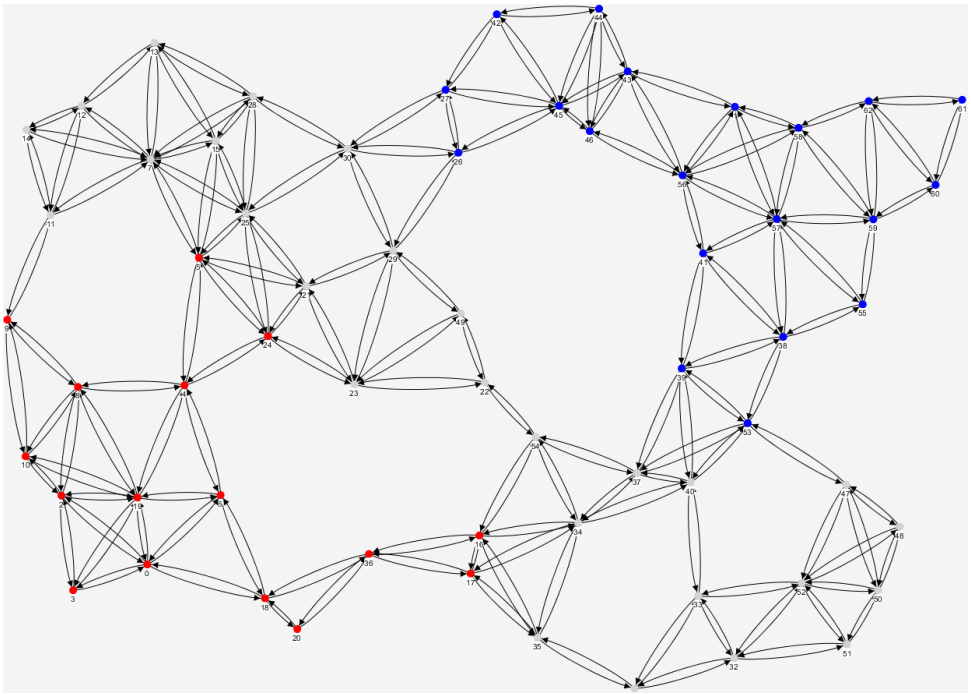


Figure 9.2: Visualization of the Routing protocol at time 3 s. Red nodes: received race number from first sink node. Blue nodes: received race number from second sink node. Lightgray nodes: Received no race number.

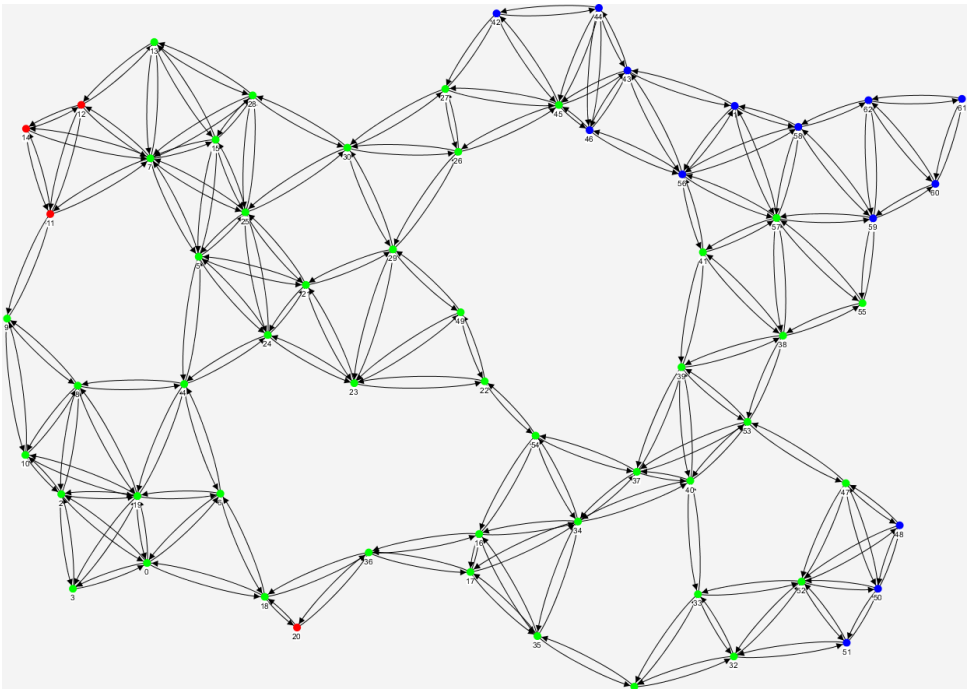


Figure 9.3: Visualization of the Routing protocol at time 5 s. Red nodes: received race number from first sink node. Blue nodes: received race number from second sink node. Green nodes: received race number from first and second sink node.

These results are only based on few simulations that we visualized, and in order to support these results we load the models in UPPAAL, and run a number of UPPAAL queries on both models, that estimate the probability of all nodes having received a race number from both sink nodes. Following is the UPPAAL query, where we run it with different time bounds x, where x is either 3, 4, 5, 6, 7 or 8 seconds converted to model time units:

```
Pr [<= x] (<> forall(i:node_id) Node(i).OUTPUT_first_sink_race  > -1 &&
                                Node(i).OUTPUT_second_sink_race > -1)
```

The results are shown in Table 9.1, which clearly shows that this small change in behavior of the protocol model will decrease the time it takes to propagate the first race number of sink nodes to the entire network. The visualization with the same VQ on the revised model can be seen in Figure 9.4.

Figure 9.4: Visualization of Experiment 9.1 which visualizes when nodes receive race numbers from sink nodes. This simulation uses the revised model. Please note that the video is shown in 50 % real time.
The QR Code will redirect to the following link:
https://youtu.be/-BiDkfUcU5g

We were not aware of this behavior when we created the model, and this experiment shows how even authors of the model can discover details about what is going on in the model with help from VisuAAL by visualizing the behavior of a model.

Time bound (s)	Original model result	Revised model result
3	$[0, 0.097]$	$[0, 0.097]$
4	$[0, 0.097]$	$[0.333, 0.433]$
5	$[0, 0.097]$	$[0.874, 0.974]$
6	$[0.334, 0.434]$	$[0.903, 1]$
7	$[0.893, 0.992]$	$[0.903, 1]$
8	$[0.903, 1]$	$[0.903, 1]$

Table 9.1: Probability estimate of all nodes receiving a race number from node 0 and node 1 within a time bound. Standard UPPAAL settings are used for running the queries.

Experiment 9.2 - Routing Protocol - Data Routing Visualization

The purpose of the Routing protocol is to route the data using a fastest path. Note that this might not be the path with the fewest links, but rather is the route where the package arrives the quickest. To obtain an heuristic of this, the race numbers are used, so that the reverse route of the fastest race number transmission is used. Further notice, that the route that the race numbers take is dependent on the internal timings in the nodes, and their transmission offsets,

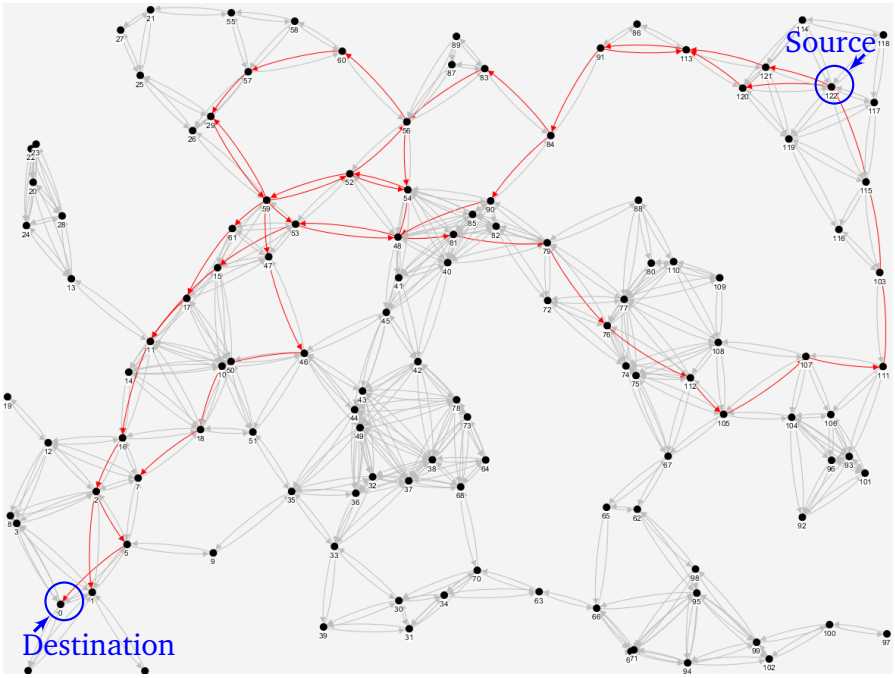


Figure 9.5: Visualization of the Routing protocol at time 20 s, with a maximum race number of 5. Edges turn red when at least one of the 5 simulations includes the edge in a route.

which are randomly selected on startup. We will now take a look at how we can visualize the route that a data package takes through the network. As an example, we will make node 122 send a single data package to sink node 0, in a topology with 123 nodes in total. To track where the data has been sent, we add the output variable `OUTPUT_sent_data_to`, that we can use to mark edges when data is sent from a node to a neighbor. In the simulation, race numbers are flooded in the network and as soon as node 122 receives a race number from node 0, it sends a data package back. In order to show paths from several simulations we use the simulation average of 5 simulations. It takes around 20 minutes to run the simulations. In Figure 9.5 we show how the data is routed for the 5 simulations. For the VQ we want to show every edge that is included in a route from node 122 to node 0 across the simulations. If an edge is part of a route in a single simulation, the output variable `OUTPUT_sent_data_to` is 1. Since we have 5 simulations making up the average simulation, the corresponding output variable will thus have the value $x/5$, where x is the number of simulations that exhibit this edge. To show all edges that are part of a route, we use the following VQ expression, where an edge will be gray if it is not used, and red otherwise:

```
[gray,red] OUTPUT_sent_data_to > 0
```

The full visualization video can be seen in Figure 9.6 and in Appendix ZIP, where it is named `routing_exp_5racenumber.sim`. In the snapshot in Figure 9.5, we can see that there is a wide variety in the routes chosen, and that there are a number of edges where the routes use both directions, which should never happen, because using such a loop will always result in a longer route in a static topology. This is especially visible in the top left corner, where the edge

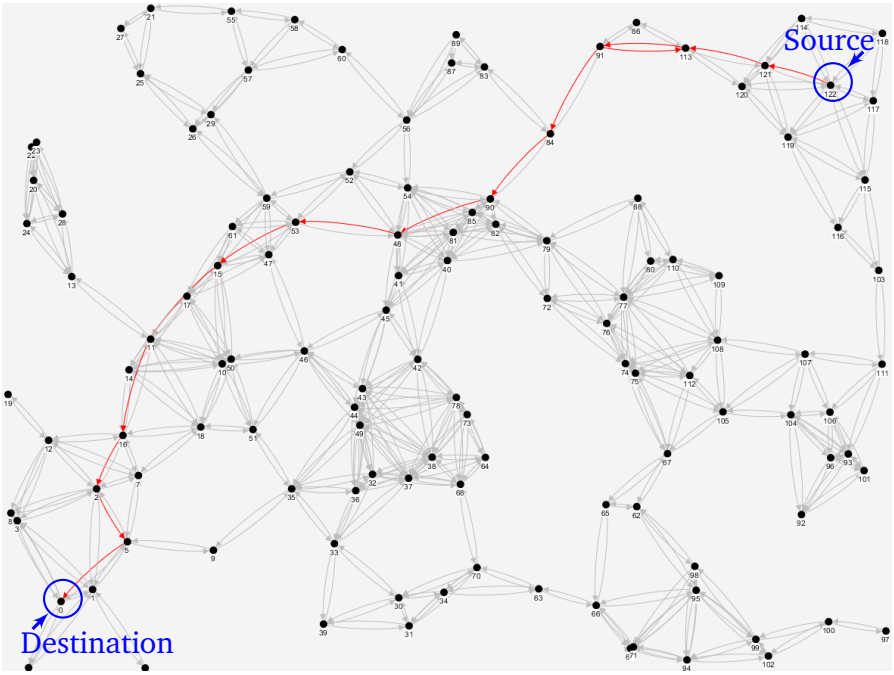


Figure 9.7: Visualization of the Routing protocol at time 20 s. One simulation with a which reaches the sink node. Edges turn red when they are included in the route towards the sink node.

between 29 and 59 has been used both directions, with two incoming edges but only a single outgoing edge.

Figure 9.6: Visualization of Experiment 9.2. Edges are colored red when they are included in one of the five simulations.
The QR Code will redirect to the following link:
`https://youtu.be/Pod2-R2w5bo`

In addition, by switching between the simulations, we can see that several of the simulations do not reach the sink node. The reason is that the maximum race number is too low for the topology used. We know that the maximum race number needs to be set at a sufficiently large value, such that a node can differ between when a race number is newer than the previous. The issue of not being able to distinguish between race numbers, and thus not sending through the optimal route, was one of the issues that we explored in [29]. Recall that the maximum race number for this experiment was 5. If we look at Figure 9.7, we can see a simulation where a route was chosen that actually reaches the sink node within the allocated 30 seconds. While this route does make a detour initially and bounces between nodes in the top, the message is received by node 0 after 20 seconds.

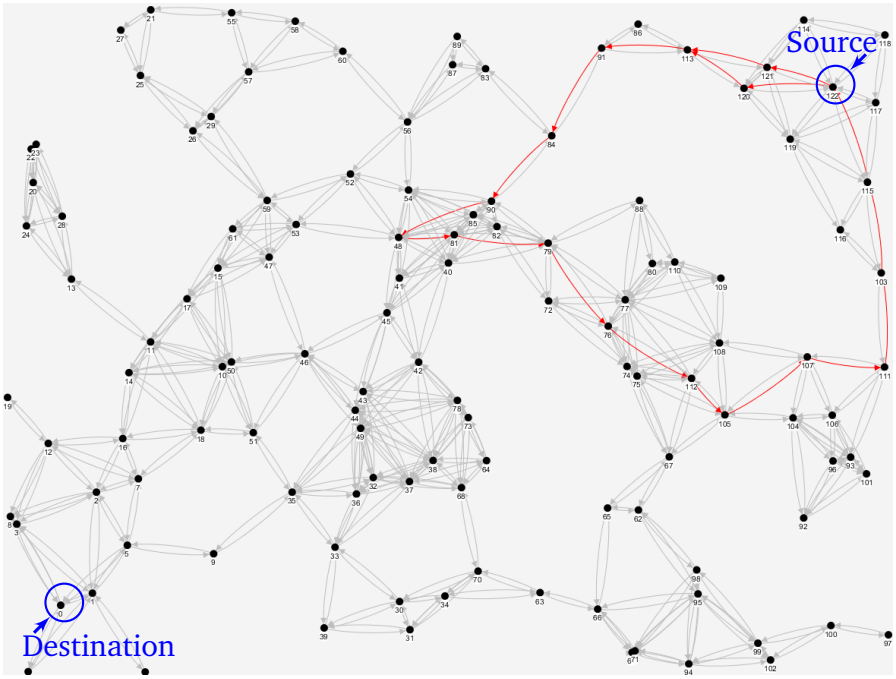


Figure 9.8: Visualization of the Routing protocol at time 20 s. One simulation with a route that does not reach the sink node. Edges turn red when they are included in the route towards the sink node.

One of the other 5 simulations that did not reach the destination within the 30 seconds is shown in Figure 9.8. Note the direction of the colored edges, which show how the package was transported around the network.

In order to show how a larger race number range affects the protocol behavior, we configure the model such that the maximum race number is 10, and run 5 new simulations. The result from this is visualized in Figure 9.10, which uses the same VQ expression. For these simulations all 5 simulations reach node 0 within the time bound, and we do not see any loops in the utilized routes. The full visualization video can be found in Figure 9.9 and the simulation file is included in Appendix ZIP, where it is named `routing_exp_10racenumber.sim`.

Figure 9.9: Visualization of data routing with 10 race numbers. Edges are colored red when they are included in one of the five simulations.
The QR Code will redirect to the following link:
`https://youtu.be/4Qsc1FrwaK0`

This experiment illustrates how we can use VisuAAL to see the concrete effects of configuring the Routing protocol model. In addition it illustrates how a specific topology might be explored, such that given a precise model, companies like Neocortec can confidently say how a given configuration will behave in a given use scenario.

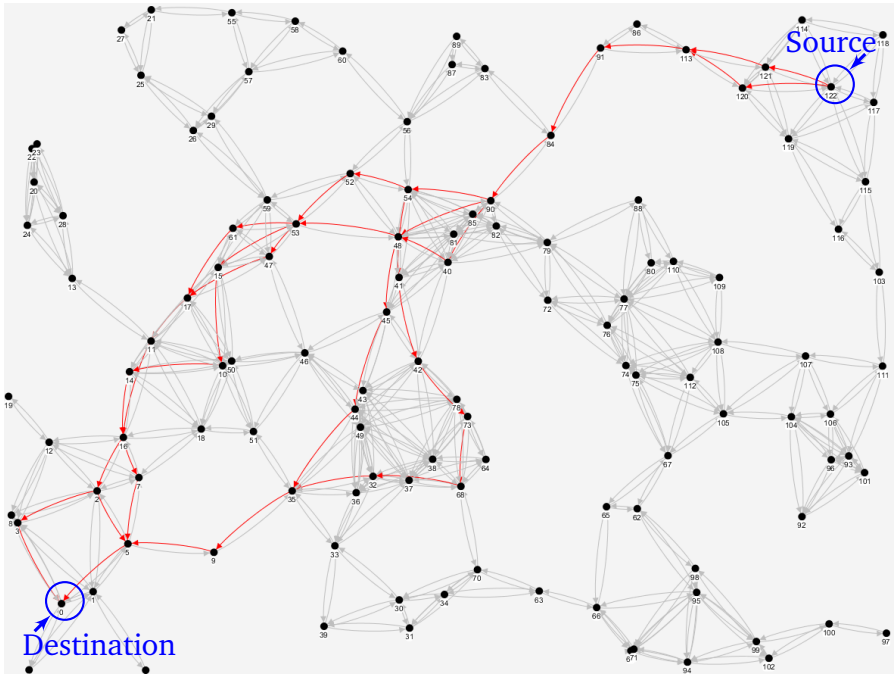


Figure 9.10: Visualization of the Routing protocol at time 20 s, with a maximum race number of 10. Edges turn red when at least one of the 5 simulations includes the edge in a route.

Experiment 9.3 - Routing Protocol - 50 Simulations

In Experiment 9.2 we discovered that if we use up to 10 race numbers, we would not generate incorrect routes towards the sink node. However, we made this conclusion on just 5 simulations. We will not be able to check that this can never happen for the given topology, since the model is far too big for standard model checking. Instead, in this experiment, we want to base that conclusion on more data. We have thus simulated 50 runs of the Routing protocol, with the same topology and configuration, and will illustrate the behavior on a larger data set. This was executed over the span of 4 hours on our Linux server. The simulation file is attached in Appendix ZIP named `routing_exp_3_50sims.sim`.

When we use a larger data set, manually inspecting all the routes to ensure that they conform becomes difficult, because they may overlap such that it appears there are incorrect routes or loops. Thus, Figure 9.11 shows the average of the simulations (as described in Section 5.9), such that we can get an overview of which edges are more often used in a route across the simulations. Figure 9.12 shows a snapshot at time 30 seconds. Note that this simulation is different from the ones shown so far, in that we are interested not only in which edges are used, but also the frequency of their use. We expect that most of the simulations follow the same route, such that a darker color is used along the direct route, with the edges close to the direct route being less frequently used. We also expect no loops in the routes obtained.

We have used two VQ's. The first is as following:

```
Node.OUTPUT_first_sink_race
```

This will change the color of a node from black to red when it has received a race number from

sink node 0. Recall that the race numbers are flooded before any messages are sent, and by visualizing the race number on the nodes, we are also able to see how the propagation of race numbers progress across the 50 simulations.

The second VQ is as following:

```
[ghostwhite, red] OUTPUT_sent_data_to
```

This will change the edges from the color “ghostwhite” to a gradient of red. It just so happens that “ghostwhite” is almost the same color as our background, rendering unused edges invisible. The more often the edge is used in a route towards the sink node 0, the more red it will appear.

Figure 9.11: 50 simulations of the Routing protocol with a 123 node topology. Nodes are colored red when they receive a race number. The more often a edge is used in a route for data transmissions, the more red it will become. Note that the video shows the simulation being played at 50% of real time.
The QR Code will redirect to the following link:
https://youtu.be/0hWEznyqTr8

We can see in the visualization how all routes from node 122 are heading towards sink node 0. The majority follow the same edges, however, there are some lightly colored edges, which show the diversity of the routes. There are no loops, and it appears that all 50 simulations managed to send the data package to the sink node within 30 seconds. Interestingly, it seems that there is consensus among the simulation about the first few edges, where almost all simulations have chosen the same edges toward the left, except for the choice between nodes 120 and 121 where the simulations seem to split between the two candidates. All simulations use node 113 after this. Now that we have 50 simulations, we can with higher confidence say that 10 race numbers is a better configuration than 5 race numbers for this topology and setup.

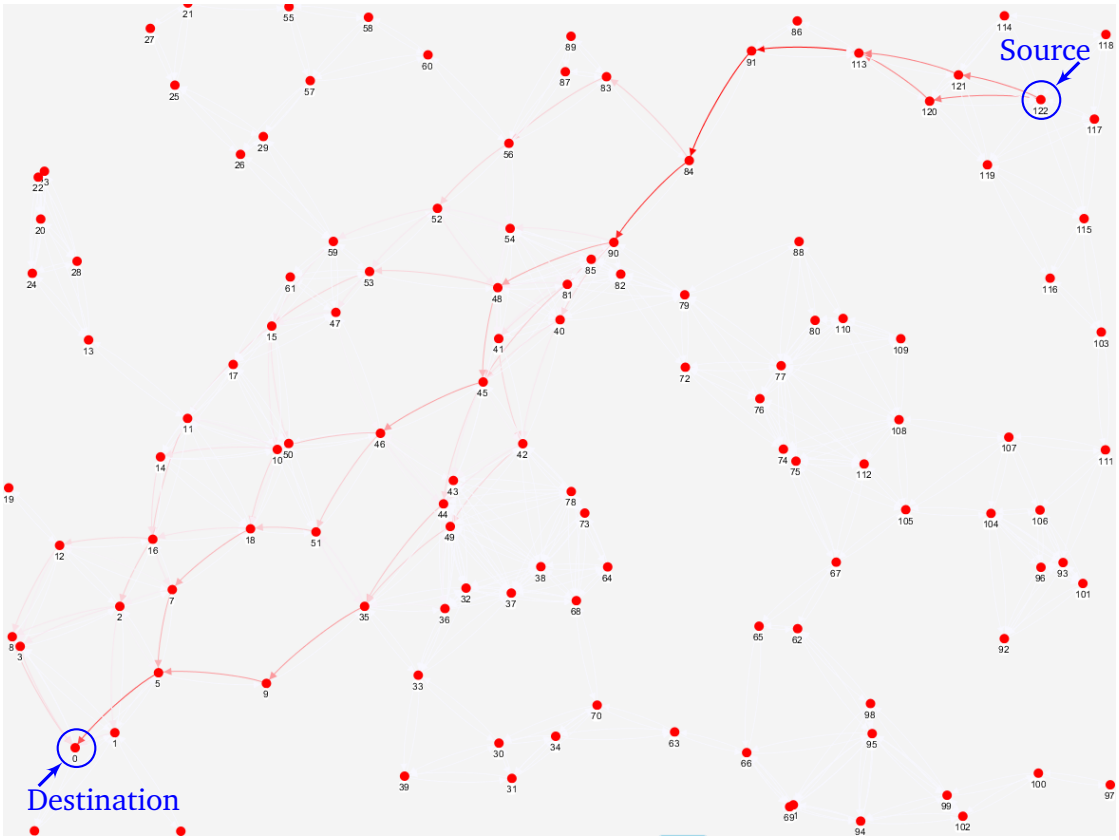


Figure 9.12: Snapshot of the simulation presented in Figure 9.11 at 30 seconds. Nodes are colored red when they receive a race number. The more often a edge is used in a route for data transmissions, the more red it will become.

10 Current Applications of VisuAAL

In this chapter we will describe the users that already are using VisuAAL or have planned to do so.

Chapter Organization This chapter is organized as follows:

- In Section 10.1 we present the collaboration with Neocortec and LinkAiders,
- in Section 10.2 we present a modified version of the MAC protocol model introduced in Chapter 8 which contains information about the battery level.

10.1 LinkAiders

Neocortec currently have a collaboration with the company LinkAiders to improve communication availability in disaster areas, specifically to allow coordinators a fast and efficient overview of the impact of disasters that may occur.

As mentioned, LinkAiders and Neocortec work toward building devices which can be used across large areas in the Philippines during natural disasters. When a disaster hits the area, the infrastructure is very often destroyed, thus normal telephones cannot be used. The device from LinkAiders, which is called *Reachi*, will help in this regard, by having the Reachis themselves work as a large scale mesh network. By providing relief workers in the affected areas with Reachis, they will be able to report the status of their area more quickly than what is currently possible, allowing an overview of the impact that the disaster caused, which enables more effective relief help. Neocortec's role is to provide the mesh networking capabilities for the Reachi using a special long range module, and thus the Reachi will use the protocols that Neocortec developed. LinkAiders develop the user interface and design all other aspects of the Reachi.

LinkAiders expect to deploy thousands of devices across the Philippines[1], where a few of them will act as sink nodes. These sink nodes have a satellite uplink, and reports directly to the coordinators, so that the disaster relief can be directed to where it is needed the most. A prototype version of the Reachi can be seen in Figure 10.1.

At the time of writing, prototypes for the Philippines project are being developed and tested. We were approached by Neocortec and LinkAiders because they are performing tests where they deploy devices and log the internally reported connectivity and locations of the devices. Neocortec informed us that it would be in their interest to visualize how the nodes have moved, and how the protocol has dealt with this change in topology. In addition, they want to simulate how their protocols would behave, based on this realistic topology, and to compare this with the observed behavior. In the prototypes, a special log file is generated for each node in regular intervals. This log file will contain information regarding the node location and neighbor connectivity. Based on these log files, we will be able to calculate a topology, annotate the

[1] Neocortec published more information here: https://gallery.mailchimp.com/0ba00e8442e65e6e68d594bb8/files/1903658f-e0fd-4d71-af9b-b04688e4fa2a/65K.pdf

Figure 10.1: Prototype version of the Reachi from LinkAiders website [33]

model when the topology changes, and also visualize the movement of the nodes. Unfortunately, the tests will not be done before we finish our project, so the following tests are only based on dummy data. More specific details of this GPS log can be found in Section 5.6.

10.1.1 Contribution to LinkAiders

Our tool is able to provide LinkAiders with an overview of how the connectivity changes over time in their test, and at the same time a tool to simulate how the protocol behaves in real-life scenarios. This will allow LinkAiders to not only visualize their test, but also to examine the effects of different configurations.

LinkAiders are initially only interested in visualizing the GPS data they obtain from their tests. Each node will log its GPS coordinates, the internal neighbor list, and the RSSI for the connection to each neighbor neighbor. When the nodes move around, the neighbor list changes, and the RSSI, which is a measure of the connectivity strength, will also change over time. Our goal is to load this in VisuAAL, and show how the topology changes over time, so that Neocortec and LinkAiders will be able to more easily visualize the network based on the logged data, and through this gain additional knowledge of the data. To our understanding, they are not initially interested in simulating the protocol behavior on top of the topology that arises from the GPS log data, and thus we need only show the topology over time with RSSI values on the edges.

To do this, we create an empty model for this purpose. The model has a single template with a single location and no edges. The only other content in the model is a topology and the required `CONFIG_NR_NODES` variable. This does not affect the number of templates, since there will be no variables attached to nodes in this model. Thus, when the dynamic updates templates are added, there will be only two templates in the model being used for the simulation, ensuring that the limiting factor in the instantiation and simulation time is the dynamic update template.

Since LinkAiders have not completed their tests at the time of writing, we generate a GPS log file with the required format, where there are 98 stationary nodes, and 2 nodes which log their location every second up to 500 seconds. The node moves 7 meters for every second in average. Node 0 starts in the bottom right corner and moves toward the top left corner, whereas node 1 starts in the bottom left corner and moves toward the top left corner. In our data, the moving nodes are connected to all neighbors within 400 meters, based on the euclidean

distance between nodes. We calculate a signal strength between neighbors, such that nodes that are close to each other have a high signal strength. This signal strength can be used as an output variable in VisuAAL. In Figure 10.2 this experiment is shown at time 222555 ms. The simulation took 10 seconds to generate on a model which has no actual protocol. The only thing that can be observed on this simulation, is the output variable `OUTPUT_RSSI`. A snapshot of the simulation can be seen in Figure 10.3, but we strongly recommend the reader to view the video to see how the topology changes. The first 50 GPS log entries from the entire log file can be found in Appendix D. The entire file with all 50100 log entries, which we used to generate the dynamic topology can be found in Appendix ZIP. The log is called `GPSLog_dynamic_0_1.txt` and the simulation file is called `DynamicPathing100Nodes500000ms.sim`.

Figure 10.2: Topology with 100 nodes, where two of them move dynamically around. The more red an edge appear, the higher signal strength.
The QR Code will redirect to the following link:
`https://youtu.be/qTRV6itAv3o`

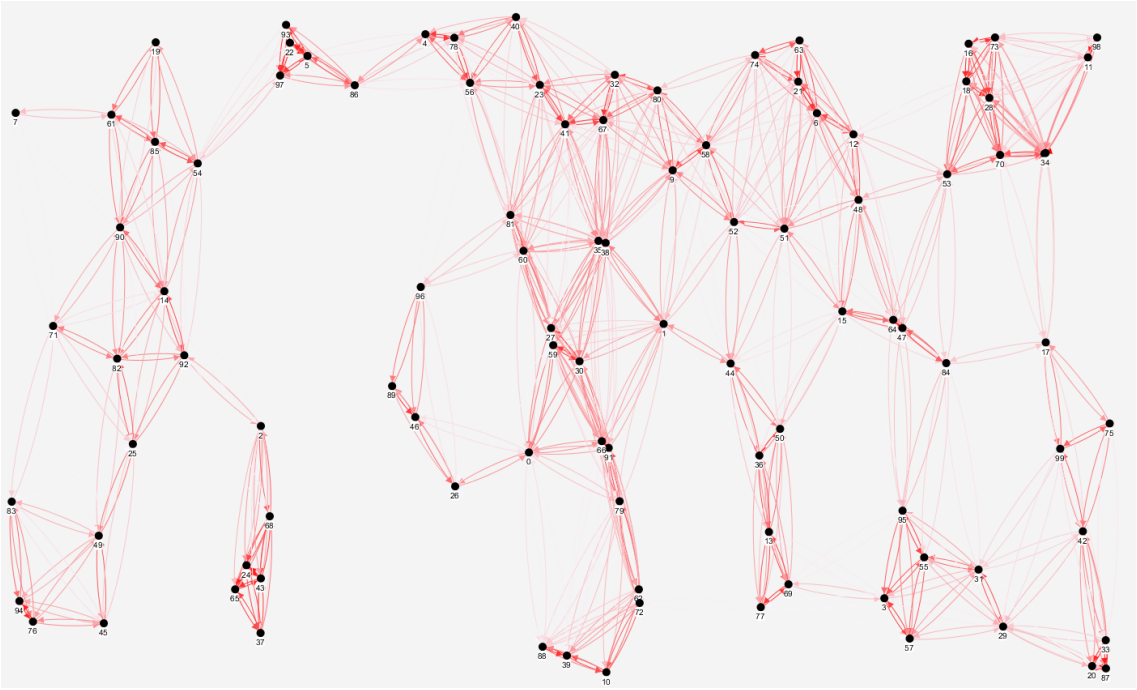


Figure 10.3: Snapshot from Figure 10.2 showing signal strength

10.1.2 Performance Data for GPS Log Loading

LinkAiders and Neocortec are interested in simulating the protocol behavior using a realistic topology recorded from a GPS log for longer periods than we typically have done in our project. In [29] we discovered that this is not trivially accomplished since it takes much time to generate

these simulations, depending on the size of the topology needed. We showed that our protocol models based on our understanding of Neocortec's MAC and Routing protocols are able to simulate data for 100 nodes at approximately the same pace as real time, thus it takes as long time to generate the simulation data, as the simulation data spans. Generating data for long periods of time is beyond the scope of this project.

Instead of attempting to generate simulations for long periods of time, we will try to stress the system in other aspects to see which limitations apply. This should give us an idea of what VisuAAL can reasonably be expected to do. In relation to the experiment LinkAiders perform, we have generated a GPS log file with 50100 log entries simulating 100 nodes, where two of them move. Each node logs its position, all neighbors and their RSSI for every 50 ms. All in all, there are 1686 dynamic updates in the file, where an edge has changed. If we include the RSSI changes and the coordinates of all nodes at all times, which are not propagated to the model, there are 453190 updates.

We are able to parse and handle the discussed log file in VisuAAL without any considerable slowdown. If we use the MAC Protocol model based on Neocortec's MAC protocol with a 100 node topology, loading, parsing and propagating the changes to the XML model is performed in a matter of seconds. Executing a query bounded at 1000 ms is done on Windows 10[2] in 75 seconds, and 5000 ms is done on Windows in 226 seconds.

To try and push the log file size further, we generate a log file with 15 nodes that are all moving, and where all nodes move to random locations at all times. Thus, the connectivity changes constantly, and there will be a huge number of dynamic updates to be propagated to the model. There are 20160 log entries in this file. This file results in 86338 locations for the dynamic update template in the model. These annotations to the model have a consequence on the run time during simulation, and an especially big impact on the memory usage for the model. The Windows version of UPPAAL is a 32-bit application [51], and in our experiments, it has been impossible for us to run simulations on a model with this many dynamic updates, before UPPAAL runs out of memory. However, the Linux version of UPPAAL is 64-bit, and here VisuAAL is able to read, parse and perform simulations with the same log file. The execution time for a simulation bound to 10000 ms on this model on the Linux server was $1,112,855$ ms, or approximately 18.5 minutes.

We executed a second query with a 1 ms time bound on the same model and machine, and found that this finished in $1,108,741$ ms, meaning that if the exact same amount of time was used to load the model for these two runs, the vast majority of time was spent doing this.

Based on this, we conclude that the main limitation in regards to reading and using GPS log files is dominated by the memory usage of the resulting model, and the model load time. It seems that the simulation execution time is not affected in as large a degree as these other factors. While Chapter 11 explores how the simulation run time can be improved further for mesh network protocols, improving the memory footprint and model load times are beyond the scope of this report.

10.1.3 Parallel Computation of Simulations

One issues with VisuAAL is that it will still take a long time to simulate the protocol when one wants to simulate multiple days or even weeks of real time of a large realistic topology. However,

[2]Intel Core i7-3610QM

once a simulation has been run, it is saved for later analysis, meaning that the simulations could be performed on a remote server, from which the results can be extracted on completion.

Since all the versions of UPPAAL and verifyta we have used are single-threaded, having a solution where verifyta is run on a remote device or through some other wrapper can also be exploited to run many simulations in parallel, and thus use all the cores on the remote server to speed up simulation. Since all the simulations are calculated independently of one another, executing a query where N simulations are needed can be emulated by having N concurrent processes execute one simulation at a time, and combining the data from each process. Using this approach comes at the added cost of having each process instantiate the model and maintain this in memory for each process. VisuAAL supports this idea of executing verifyta in parallel in the sense that one can start many verifyta processes through VisuAAL, running in parallel with different models and parameters at the same time. When a process finishes, the data processing is also performed concurrently, and a tab is opened with the results. Thus, VisuAAL can be used to have many experiments running concurrently. We do however not support combining the data from separate processes, but this could easily be extended in the future.

One issue that must be overcome in this regard is the verifyta initialization time and closely related, the memory consumption. When UPPAAL executes queries, it will take the model specification and compile this into a format in memory that can be used for running queries. Once this is done, the query can be executed. In our case the verifyta initialization is never finished on the 32 bit Windows version of UPPAAL, since we exhaust the addressable memory allocated to the verifyta process. For simulations, this same model is used to execute all the simulations, meaning that the setup time is independent of the number and length of simulations that must be executed. This is important when we want to consider parallelization, where the model is shared among the simulations being executed.

If the simulation execution time is dominated by the verifyta initialization time, having each process construct and maintain their own model is problematic, but if not, parallelization can be achieved with little additional work. The verifyta initialization time largely depends on the model. For simple models and sufficiently long simulation bounds, it might make sense to construct a model for each process, but in the concrete case examined in Section 10.1.2, verifyta initialization took $\approx 99.996\%$ of the total query execution time, and in this case sharing the model and executing the queries sequentially will be far superior to having a model for each query.

10.2 Energy Consumption

We introduced the MAC protocol in Chapter 8 and the model of it that we created in [29]. Ivanov [25] has as part of his student project added energy consumption to the model, based on the kinetic battery model [34]. One issue that Ivanov discovered, is that it is very time consuming to simulate a model that includes the kinetic battery model. Instead Ivanov annotated the MAC protocol model with the current draw for nodes that should be expected based on the state of the model, i.e. if it is transmitting or sleeping. He then does post-processing based on the current draw of nodes, and computes the analytical solution of the energy model. The post processing is performed in an application that wraps verifyta. The wrapper has the same output format as verifyta, such that it can be used by VisuAAL directly.

We will now briefly explain how the kinetic battery model works, based on [34]. The kinetic

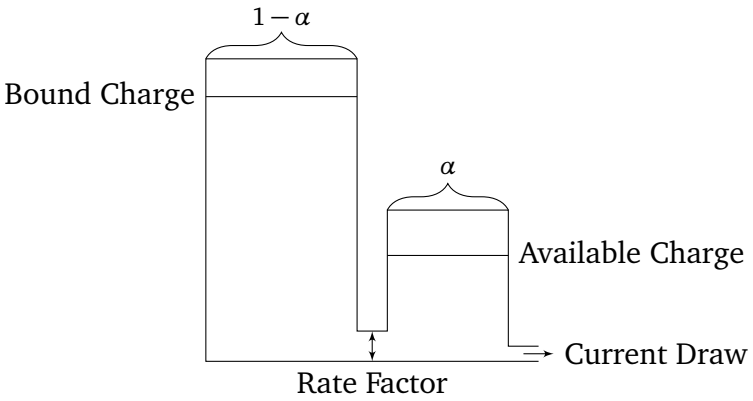


Figure 10.4: Illustration of a battery where α is a number between 0 and 1, describing the split between available and bound charge

battery model, models a battery as having an available charge and a bound charge. We can always draw energy from the available charge, but cannot draw energy directly from the bound charge. When the available charge is reduced however, energy from the bound charge will slowly flow to the available charge. If a lot energy is drained over a short period of time, it could drain the available charge completely. Instead the energy usage should be balanced, in order to use the bound charge efficiently. Ivanov [25] calculates the available and bound charge in the wrapper and outputs it appended to the UPPAAL output.

An overview of the Kinetic Battery Model is shown in Figure 10.4. Note that the flow of energy between the bound and available charge is dependent on the Rate Factor, the capacity of the battery and the ratio of capacity in the bound and available charge.

We will now visualize the energy usage of nodes and how the available charge behave during simulation. We have from the output of Ivanov's wrapper three output variables:

`OUTPUT_battery_i` The current draw

`OUTPUT_battery_a` The available charge

`OUTPUT_battery_b` The bound charge

10.2.1 Battery Charge Level

In this experiment, we look at the energy usage for nodes in the MAC protocol. We setup the kinetic battery model for nodes to follow the specifications of the VARTA CR2450 battery provided by Ivanov [25]: The nodes have approximately a total charge of 2.47×10^9 mAms, where 5.3 % of the charge is the initial available charge and a rate factor of $^{3.34\times10^{-8}}/_{ms}$.

The specifics of how the energy flows and how fast, is out of the scope of the project, and it is sufficient to know that energy does flow over time at a rate dependent on the ratio between the bound and available charge and the capacity in each. For more information see [25, 34].

We configure the model to have a 20 mA current draw when transmitting or receiving and a current draw when sleeping of 0.5 µA. We are not claiming that this is a perfect annotation to show precise energy consumption, but these values are based on data from the CC1110

transceiver datasheet [47]. We have come to understand that Neocortec use a transceiver very similar to this. Note that there are many factors to consider for the current draw for different operating modes and thus realistically there is not only one value for e.g. transmitting.

For this experiment we expect the nodes that have many neighbors to use the most energy, because they wake up to listen for neighbor data transmissions often. Due to the complex relationship between the bound charge and the available charge, the available charge may over time either increase, decrease or stay balanced. For instance if the protocol models nodes has a balanced energy usage, we expect that the nodes will draw sufficient amount of energy, that is transferred from the bound charge, and the full charge of the battery will be used, without draining the available charge completely.

The intention was to do this experiment for 100 simulations, that each run to 3600000 model time, which equals 1 hour. We ran the simulations for approximately 12 hours, and it provided a lot of data. Afterwards VisuAAL should collect the data from verifyta, but an exception occurred. We have not executed such a large experiment before, and did not use an efficient way to store the many gigabytes of textual data we received from verifyta. We think this can be easily fixed, however due to time constraints we cannot rerun the experiment for 100 simulations.

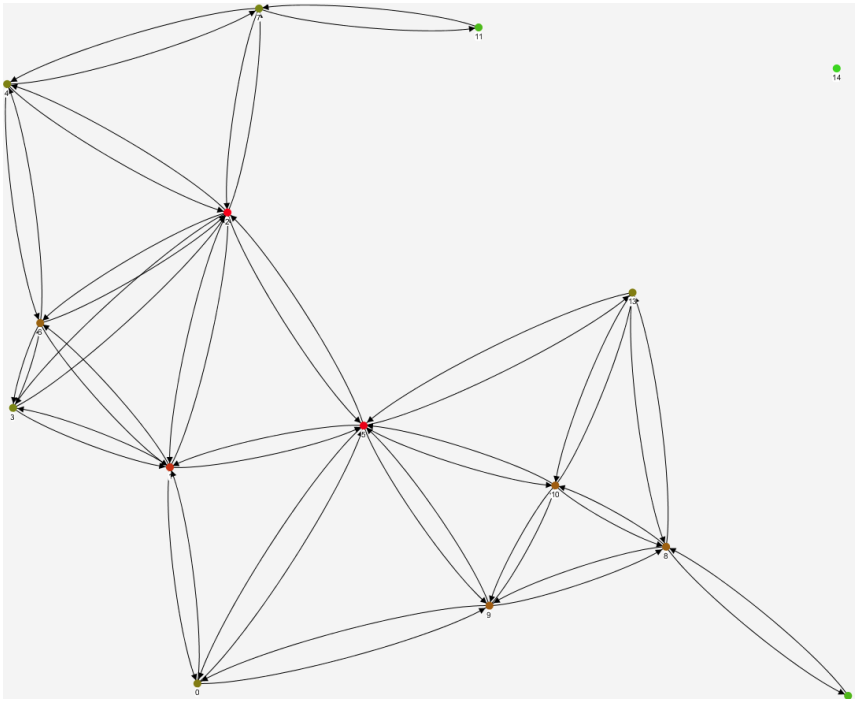


Figure 10.5: Snapshot of the average simulation at time 1 hour. The shown VQ expression shows a gradient between red and green based on the available charge of nodes.

Instead we look at the average of 10 simulations, which took about an hour to simulate. In Figure 10.5 we present a snapshot at time 1 hour. We use the following VQ expression, which shows the available charge for the nodes. We use a gradient where the lower bound that is red, is the lowest available charge of all nodes at time 1 hour and the upper bound is green, which is the initial available charge of nodes.

```
[red: 109900000, green:131800000] OUTPUT_battery_a
```

The full simulation is shown in Figure 10.6 and the simulation is available in Appendix ZIP where it is named `energy_10.sim`.

Figure 10.6: Video of the average simulation for 1 hour, running in 120 times real time. The shown VQ expression shows a gradient between red and green based on the available charge of nodes.
The QR Code will redirect to the following link:
`https://youtu.be/hGfMww97xWw`

As expected, we can see from the results that the nodes with few neighbors uses little energy compared to the nodes with many neighbors. For instance node 14 that has no neighbors, used the least amount of energy and node 5 which has 6 neighbors used the most amount of energy. In Table 10.1 and in Figure 10.7 we show the difference of charge levels for nodes compared to the initial available charge level. These differences we can use to get an idea of how fast the nodes will run out of energy, where in a continued simulation, we expect the nodes would run out of energy at different times.

We are unable to conclude from this experiment whether there are other hidden parameters that affect the battery drain, such as the activity of second order neighbors, but we can conclude that there seems to be a trend that nodes with more neighbors use more of their available charge.

The result thus shows interesting information about how the amount of neighbors affects the available charge level of nodes. Note that in order to run simulations with energy usage as we presented in this section, the wrapper created by Ivanov [25] is needed, which we do not include in the appendix. The simulations that we included in Appendix ZIP however, can be loaded to visualize the simulations that we ran for our experiment.

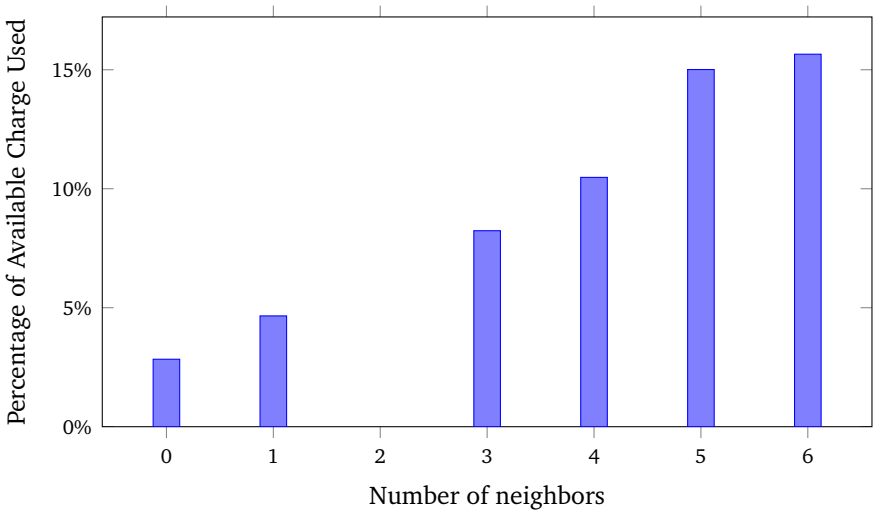


Figure 10.7: Average percentage of the available charge used for nodes with different amount of neighbors. Note that there were no nodes with 2 neighbors in the topology.

Node id	# Number of Neighbors	Available Charge / Initial Available Charge
5	6	$^{1.112E+8}/_{1.32E+8} \approx 84.35\%$
7	3	$^{1.214E+8}/_{1.32E+8} \approx 92.08\%$
14	0	$^{1.281E+8}/_{1.32E+8} \approx 97.16\%$

Table 10.1: Sampled available charge level of a few nodes relative to the initial available charge, at time 1 hour

11 Model Scalability Experiments

VisuAAL's run time is directly dependent on UPPAAL and the scalability of the models that are used. In [29] we discovered that the models we created did not scale very well in terms of simulation run time with the number of nodes. In this chapter we will do a number of experiments to find out why and how we can avoid the scalability issues.

Chapter Organization This chapter is organized as follows:

- In Experiment 11.1 we experiment with scalability for a model of a standard approach, that instantiates 2 templates for each nodes,
- in Experiment 11.2 we try to eliminate factors that causes long run time, in this case we look at removing channel synchronization,
- in Experiment 11.3 we try to eliminate the factor of many node instantiations,
- in Experiment 11.4 we introduce UPPAAL execution order in the model,
- in Experiment 11.5 we add execution order by having asynchronous nodes,
- in Experiment 11.6 we model all nodes through two templates in addition to adding execution order,
- in Experiment 11.7 we compare an optimized Slotted ALOHA protocol model to the unoptimized version from Section 3.5.

Intuitively when a model is simulated, the time it takes for a single node to perform its actions should not be affected by the size of the topology as long as the node has the same amount of neighbors. If this is the case for a model, then when adding more nodes to the topology without increasing the average number of neighbors for nodes, the simulation run time should increase linearly with the number of nodes in the topology. In [29] we presented two models of distributed network protocols, where we show that the models do not scale linearly for simulations. To gather more information about the issue, we perform a number of experiments to see how UPPAAL behaves in various scenarios with hundreds of nodes. Note that the following experiments are run on a laptop[1]. All the experiments are run 10 times for 1000 time units. The reason for running several simulations is that there is a small overhead when initializing the models in UPPAAL, which we try to minimize. We cannot see how long the initialization takes out of the total simulation run time. It is due to time constraints that we only do 10 simulations for every experiment, but increasing this would minimize the overhead even more.

[1] Intel Core i7-3630QM 2.4GHz processor, 8 GB RAM

Experiment 11.1 - Two Templates for Each Node

(a) Sender **(b)** Receiver

Figure 11.1: Templates used in Experiment 11.1. Sender and receiver templates that each is instantiated once for each node, with a node id as input.

In Figure 11.1 two UPPAAL templates are shown for the model that we use in this experiment. There is a sender and a receiver for every node. For every node the sender synchronizes with the corresponding receiver every 2 model time unit. The channels used are global and the clock is local for each node. Each node in the model is unaffected by the other nodes, and adding more nodes should not change other nodes' behavior. The model is simplified a lot in terms of neighbor connections, but it is sufficient for the experiment, since the requirement is that all nodes have the same number of neighbors for all simulations.

We now look at the simulation run time when we scale the model up to 500 nodes. In Figure 11.2 these experiment results are presented. The plot with filled circular marks is the results for this experiment. We show how long it takes to simulate the model with different number of nodes. The filled circular plot clearly shows that this model does not scale linearly with the number of nodes. Instead it has a tendency that looks like a quadratic scaling.

Experiment 11.2 - Removing Communication between Templates

To further examine the scaling issue shown in Experiment 11.1, we do an experiment with no channel synchronizations. We do this as to eliminate factors that might be responsible for the scaling issue. We remove the receiver template from the model of Figure 11.1 and remove the `send` channels, and let every node only reset its own clock every 2 time unit. Each template instantiation is now completely independent.

The plot with square marks in Figure 11.2 shows the simulation run time scaling for this model. The tendency of the data is the same as the previous experiment, however they differ with what seems to be a linear factor. The scaling issue however is still present, but the result gives us a clue; that the problem might be the instantiation of templates for each node.

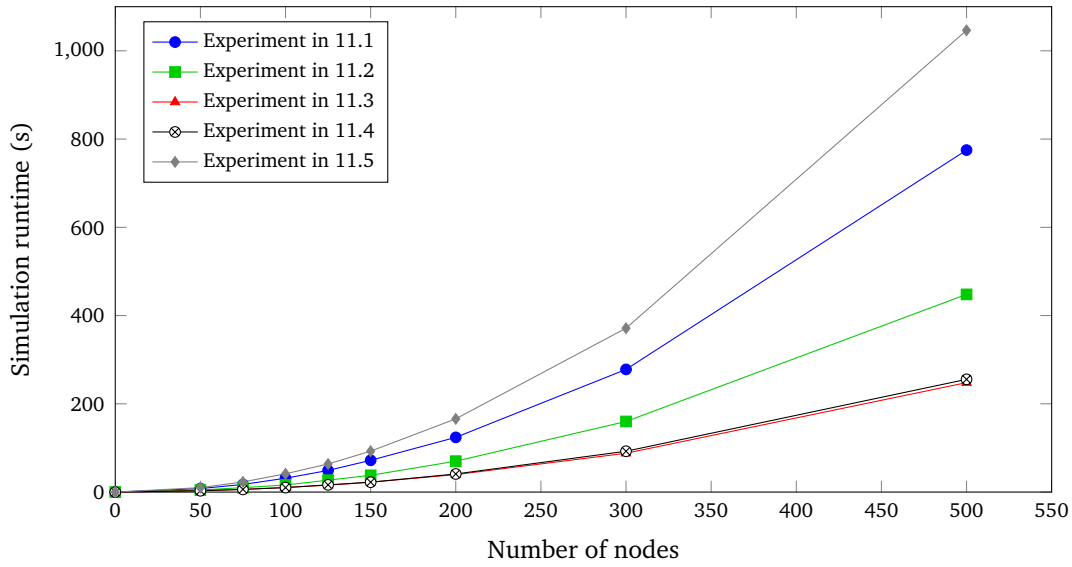
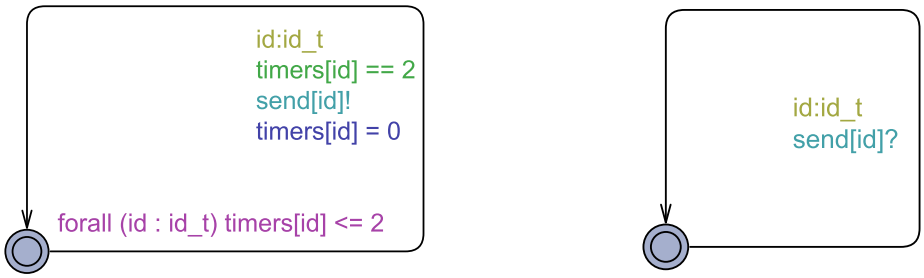


Figure 11.2: Results for Experiment 11.1, Experiment 11.2, Experiment 11.3, Experiment 11.4 and Experiment 11.5. Note that results for Experiment 11.3 and Experiment 11.4 is almost on top of each other. For all data points, 10 simulations are run for 1000 model time units.

Experiment 11.3 - Multiple Nodes in a Constant Number of Templates



(a) Sender **(b)** Receiver

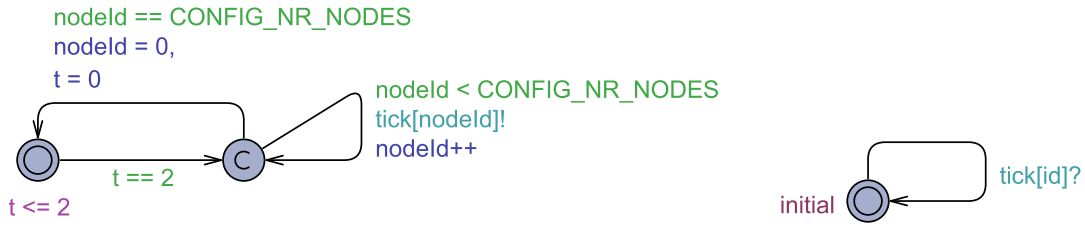
Figure 11.3: Templates used in Experiment 11.3. Sender and receiver templates that handles all nodes actions.

In this experiment we create a model with a constant number of templates being instantiated. In Figure 11.3 we show the templates for the model. The model mimics the behavior of the model from Experiment 11.1, but uses selects to do the actions of all nodes alternately. `id_t` is the integer type for node id's and `timers` is an array of clocks for all nodes. The sender has an invariant that requires all nodes to have their corresponding clocks less than or equal to 2. The edge then has a select over all node ids, and the nodes with a timer equal to 2 then synchronizes on their `send` channel with the receiver alternately. The receiver has a select over all node ids, and synchronizes over the selected `send` channel.

In Figure 11.2 the plot with triangular marks is the result for this experiment. It clearly performs better than the model with many instantiated templates, but it still does not scale linearly. From this experiment we know that it is not necessarily the many instantiated nodes that is the problem.

Experiment 11.4 - Incorporating Execution Order in the Model

The models from the first 3 experiments have a thing in common; at every 2 time unit, UPPAAL has to choose in which order the nodes progress. If n is the number of nodes, and if UPPAAL considers all possible states before progressing a node, then to let all nodes progress it will take $n+(n-1)+\cdots+2+1$ time, which is $O(n^2)$ run time. In this experiment we look into how we can progress all nodes in linear time. We expect this is possible by adding an order of when the nodes are progressed, so that UPPAAL for every node that should be progressed, only has a single choice.



(a) Scheduler **(b)** Node

Figure 11.4: Templates used in Experiment 11.4. The scheduler template that progresses and the node template that synchronizes to the scheduler.

In Figure 11.4 we show two templates for a model, where the scheduler template acts as a discrete clock for the nodes. At every 2 model time unit, it tells every node to progress. The nodes cannot have individual clocks with invariants and guards, if they are required to progress when a scheduler tells it to, because we will have issues with independent progress. With this approach however, the schedule of nodes is completely controlled by the scheduler template, as they are forced to synchronize with it.

In Figure 11.2 the plot with marks of circles with a cross within, shows the simulation run time for the experiment is shown. This experiment also did not run in linear time, and it adds to the presumption that we cannot instantiate a template for every node and get linear scaling.

Experiment 11.5 - Execution Order With Asynchronous Nodes

Before we discard the idea of instantiating a template for every node we have this experiment with asynchronous nodes. This is another idea for how node actions can be sequentialized. To induce the order we let the nodes purposefully be out of sync. If all nodes for instance can have up to 1 model time unit offset in time compared to other nodes and with the same interval between when they perform actions, this imposes an order for the nodes actions that UPPAAL will follow. This mirrors real devices in that the nodes would not be able to be 100% synchronized here due to manufacturing differences, clock drifts and other factors. Using this

idea to ensure sequential order between the nodes in a model could potentially reduce the simulation run time in addition to adding details to the model.

To test this, we use the same model as in Experiment 11.1, except that we added a new location for the sender. This new location lets the node wait up to 1 model time unit and then starts the cycle as in the original model. This initial delay will likely be different for each node, since they select from the uniform distribution between 0 and 1 model time units. Since the original cycle maintains exactly 2 model time units, this offset will be retained throughout the lifetime of the model. The receiver template is the same as in the original model. The updated sender is shown in Figure 11.5. In Figure 11.2 the diamond marked plot is the result of this experiment. This approach unfortunately still has quadratic scaling with the number of nodes, even though the nodes' actions are ordered. The experiments point towards the need of modeling all nodes in a constant number of templates to improve the scalability.

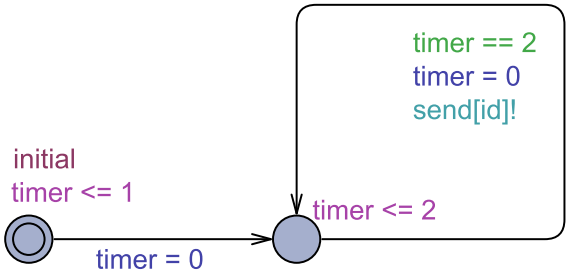


Figure 11.5: Template used in Experiment 11.5. Sender template, one instantiated for each node. The nodes are not synchronized, thus UPPAAL will progress the nodes in order.

Experiment 11.6 - Execution Order for Multiple Nodes in a Constant Number of Templates

In Experiment 11.3 we learned that we can reduce the run time by modeling all nodes as a constant number of templates. In this experiment we will model all nodes in two templates in addition to ordering the node's actions. The templates for this experiment are shown in Figure 11.6.

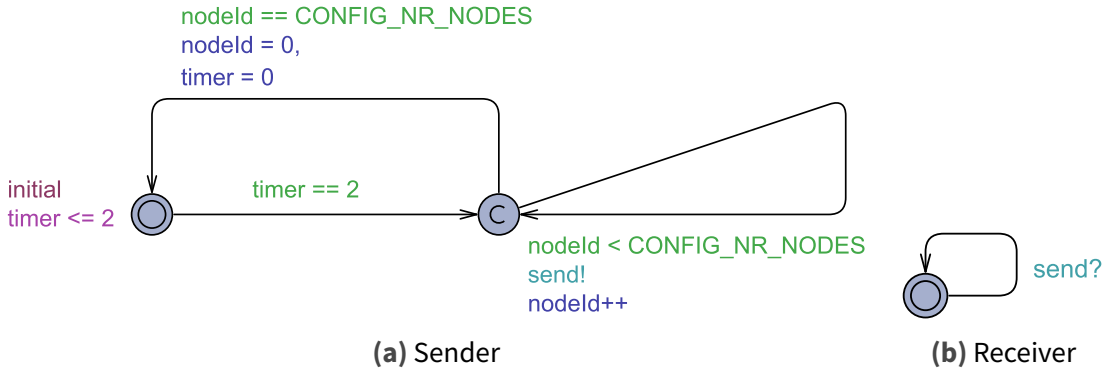


Figure 11.6: Templates used in Experiment 11.6. Two templates handling every node, where the nodes actions are ordered

Whenever the clock `timer` reaches exactly 2, the sender template moves to the next location, where all nodes are progressed in order, to synchronize with the receiver on the `send` channel. The results of simulating the model 10 times for 1000 model time units, can be seen in Figure 11.7. For this approach we get linear scaling, and we could even run 1000 nodes in just 10 seconds. Note that in this small example, the nodes only synchronize on the `send` channel, but in a real protocol we would need to manage neighbor connections based on a topology. If we can do this in constant time, this method can be useful, and we will look further into this in the next experiment.

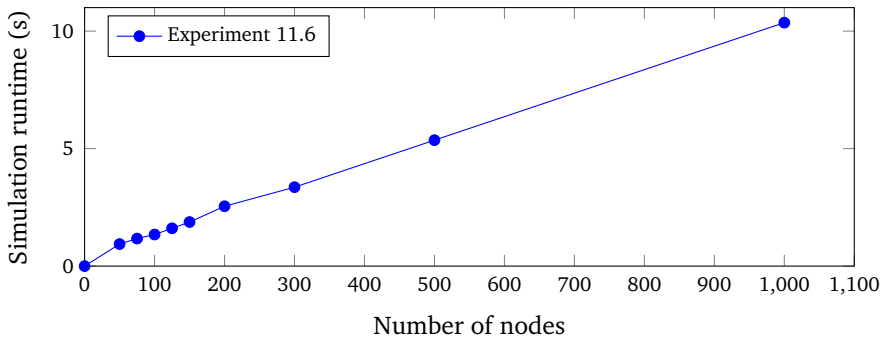


Figure 11.7: Result for Experiment 11.6. Simulations of a simple model that is sequentialized. For the experiment, 10 simulations are run for 1000 model time units, for each data point.

Even though this approach scales very well compared to the other approaches, it does set

rather restrictive limits to the model. In the simple example shown, every node can perform an action every 2 second. However if we want the nodes to have individual clocks and schedules, as is often the case for distributed network models, it becomes hard to create a model with this approach, for the following reasons:

- We cannot instantiate a template for every node, since we have not been able to achieve linear scaling in any experiment where the number of templates scales with the number of nodes.

- We cannot use an invariant on all the clocks and a select over the number of nodes as in Experiment 11.3. For all nodes to progress it must take at most linear time. A select over the number of node ids does not have an order, and is thus quadratic in time. In addition by using the invariant over all clocks, we need to know which node that causes us to leave the location, however we cannot do this in constant time, because we need either the select or to iterate over the nodes.

- We do not know of a way of letting nodes perform actions in more complex orders, for instance if a node performs a task on a schedule with a different period than that of other nodes, which causes different interleavings of node actions each time the actions must be performed.

These findings may not hold in the general case, but we have not been able to find a way to obtain linear scaling without these restrictions.

Experiment 11.7 - Slotted ALOHA

In order to see if we can use the approaches described in the previous experiment to create a UPPAAL model of a simple protocol, we have created an improved model, in terms of simulation run time, that is based on the model presented in Section 3.5 which is a Slotted ALOHA protocol model [45].

In Figure 11.8 we show the sender template of the model and in Figure 11.9 we show the receiver, which both have a single instantiation, that covers all nodes. The sender starts by choosing a random slot for each node. It then proceeds to iterate over all the slots, which is 8 in this experiment. In each slot we iterate over all nodes, and if the slot of the active node is chosen it synchronizes on the `send` channel with the node's id. It should take linear time to complete a period with the 8 slots, since we iterate over all nodes once at the start, and then once for every slot, where the nodes' actions are ordered.

The receiver waits to synchronize on a channel either when a slot is done or a node sends a message. When a message is sent, the receiver iterates over all neighbor id's of the sender node. It then tracks how many times a node receives a message in a slot, to account for collisions.

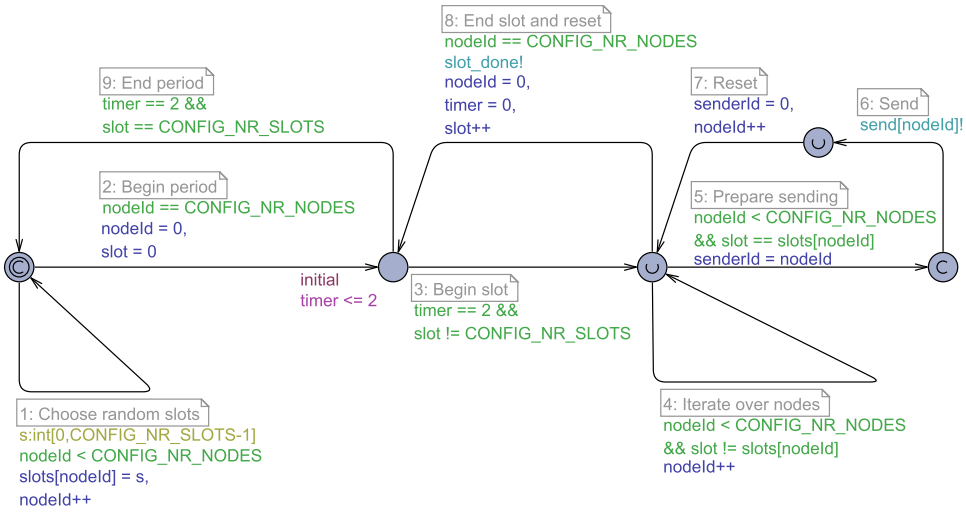


Figure 11.8: Sender template for the improved Slotted ALOHA protocol model

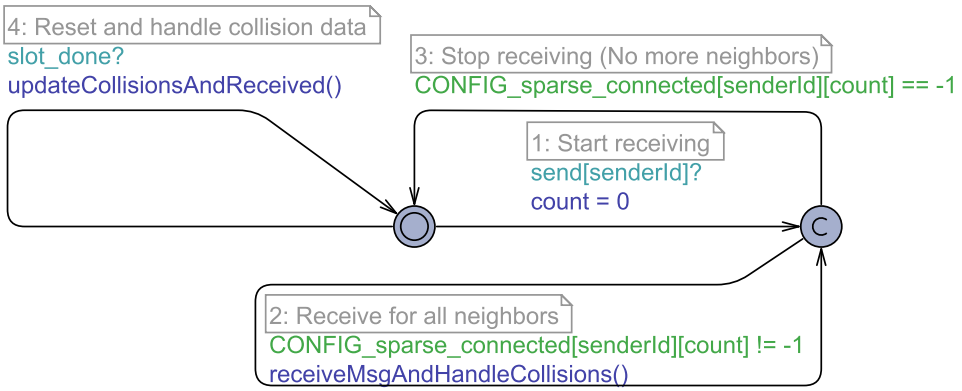


Figure 11.9: Receiver template for the improved slotted ALOHA protocol model

In this experiment we need to let nodes send a message to all neighbors in constant time, if we are to scale linearly in simulation run time with the number of nodes. To do this we created a sparse matrix for the neighbor connections. Whenever a node sends a message, it will take constant time to send to all neighbors, as long as we have a limit of the number of neighbors. For this experiment all nodes have 5 neighbors, and it will take only 5 iterations whenever a node sends.

Following we will compare the results of the improved Slotted ALOHA protocol model to the original Slotted ALOHA protocol model presented in Section 3.5, in order to gain an idea of how these techniques might affect an actual protocol model. The original Slotted ALOHA model does not use any of the modeling techniques that we have just presented for optimization. Thus, the original model instantiates a receiver and a sender template for all nodes, does not sequentialize the flow and does not use a sparse matrix.

In Figure 11.10 and Figure 11.11 we show the simulation run time for the improved Slotted ALOHA protocol model compared to the run time of the original Slotted ALOHA protocol model. We show two different scales, since there is such a huge different in simulation run time. Additionally the results are shown in Table 11.1.

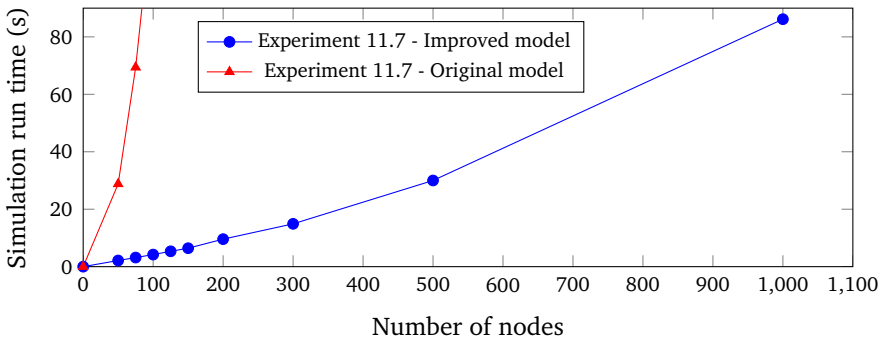


Figure 11.10: Simulation run time for the improved Slotted ALOHA model compared to the original Slotted ALOHA model. Note the Y-axis goes to 90 seconds. For both models, 10 simulations are run for 1000 model time units, for each data point.

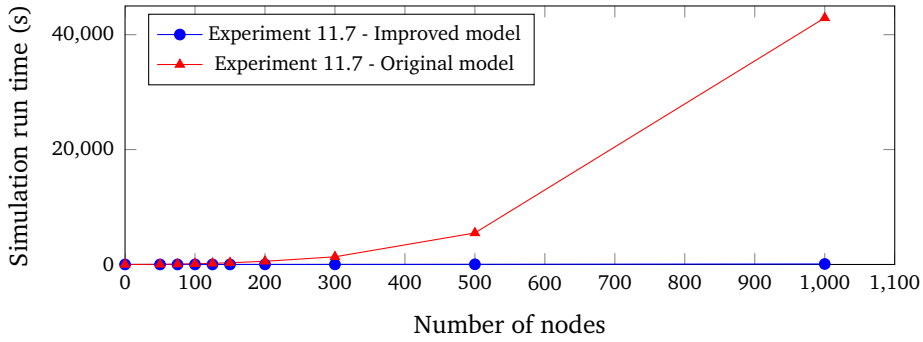


Figure 11.11: Simulation run time for the improved Slotted ALOHA model compared to the original Slotted ALOHA model. Note the Y-axis goes to 45000 seconds. For both models, 10 simulations are run for 1000 model time units, for each data point.

Number of nodes	Improved (s)	Original (s)	Ratio
50	2.1	28.8	13.5
75	3.2	69.4	21.9
100	4.2	127.2	30.3
125	5.3	201.4	37.7
150	6.4	301.1	46.9
200	9.6	554.5	57.9
300	14.9	1334.6	89.5
500	30.0	5487.6	182.9
1000	86.2	42922.2	497.9

Table 11.1: Simulation run time for the improved Slotted ALOHA model compared to the original Slotted ALOHA model. For both models, 10 simulations are run for 1000 model time units, for each data point.

Received messages	>150 messages	>160 messages
Probability - Original	[0.80, 0.91]	[0.41, 0.51]
Probability - Improved	[0.78, 0.88]	[0.43, 0.53]

Table 11.2: The probability of node 0 receiving more than 150 and 160 messages within 1000 model time units, for both Slotted ALOHA models. The models uses 8 slots with 50 nodes and each node has the next 5 nodes as neighbors. The queries are run with default statistical parameters in UPPAAL.

The simulation run time for the improved model is not linear, but compared to the original version, it is much faster to simulate. The results clearly show that the optimizations make a huge difference in terms of simulation run time, and for 1000 nodes we reduce the simulation run time by almost a factor 500. Even though both models are simple, we show a number of ways that improve models allowing us to simulate them much faster. The approach unfortunately sets limits to the model, but in cases like the one shown in this experiment, the modeling task is doable.

To show that the models are comparable to some degree we run a probability query in UPPAAL for both of the models, comparing the number of messages that node 0 receives in the two models. This only goes part of the way toward showing that the models are equivalent, but it gives an indication. Due to time constraints, we will not examine whether they are equivalent.

For the comparison, we estimate the probability of node 0 receiving a total of 150 and 160 messages withing 1000 model time units, for both the models. In Table 11.2 the estimated probabilities are shown. The data shows that the amount of messages received by node 0 is roughly the same for the two models. We do not expect completely identical results, since we use statistical model checking.

In conclusion, when modeling if we avoid situations where UPPAAL can do actions for

multiple nodes simultaneously by sequentializing the model behavior, we can greatly decrease the simulation run time. Additionally we can use a sparse matrix for neighbors, that will give performance improvements for topologies with few neighbors.

12 Evaluation

In this chapter we conclude on our project and look at what work that further would be relevant to work with.

Chapter Organization This chapter is organized as follows:

- In Section 12.1 we conclude on our project; VisuAAL, experiments and results,
- in Section 12.2 we describe the work that further could be done for our project.

12.1 Conclusion

In a world where IoT becomes more integrated in our every day life, it follows that there will be increasing amounts of wireless devices that needs to be connected. For these wireless devices, we need protocols that facilitates correct and efficient communication of data between nodes. In this project we do not look at creating these protocol ourselves, instead we want to help developers in the domain specific area of distributed networking protocols, to easily gain knowledge and overview of their protocols through visualization.

We created an application that visualizes the behavior of distributed network protocol models. Our application, named VisuAAL, uses UPPAAL as the engine to run simulations and a UPPAAL model of the protocol that is to be observed. We set a number of requirements to the UPPAAL model, that works as an interface. An example of a requirement is prefixes of variables, so that we are able to recognize them, in order to configure and use them for output data.

VisuAAL lets users configure their model, generate dynamic topologies for the model and visualize the simulations created by UPPAAL. It is essential to be able to configure the model, since it allows the user to compare different versions of a protocol. This is useful since protocols may dictate different behavior for nodes under different topologies and topology sizes. Through VisuAAL, we can generate random topologies and inject them in the model.

In the real world, nodes move and thus the connectivity between nodes can change. We support this behavior by being able to dynamically change topologies and other variables during simulations. This we do by updating variables in the model at specific time steps. This functionality we also use to load location and connectivity data from real devices, and use this in the model when simulating the protocol model.

When we have a model that is configured, we run simulations with UPPAAL and use the output to visualize the protocol model behavior. VisuAAL visualizes the topology with nodes and edges from the simulations. On this topology different types of data can be shown, based on the chosen output variables. The user can with the query language VisuAAL Query (VQ), color nodes and edges based on output variables of the model. The VQ language, allows users to define how data is presented for the nodes and edges and can for instance be used to visualize abnormalities.

Case Studies

We did a number of case studies in the project, to see what and how we can visualize different protocol models. The LMAC protocol is a MAC protocol that utilizes frames. For a node, the goal is to select a frame wherein it can transmit data without colliding with neighbor nodes. For the LMAC protocol model developed by Fehnker, Hoesel, and Mader [12], we visualized the state of the nodes and showed how the number of frames affect the protocol model's behavior. With few frames in a dense network, we can see that the nodes will collide often and will rarely settle for a specific frame. If we then increase the number of frames, we can see how all nodes selects and successfully keeps a frame to transmit data in. If we know in which topology the protocol is being used, we can easily select a fitting number of frames and see how this change affects the behavior. We presented a visualization of LMAC protocol model to one of the authors of the model, who responded with the following:

> I showed the visualizations yesterday to Angelika Mader, one of the co-authors who also happens to share a course with me this semester, and we were quite impressed. It took us a while to recall how LMAC worked, but the visualization actually helped.
>
> *Ansgar Fehnker [10]*

AODVv2 is an ad hoc routing protocol. Höfner and McIver [23] developed a model of this protocol which we visualized. We focused on the queue of messages that nodes have. If the queue over time becomes large, there is too much traffic for the node and it is unable to keep up. We created a topology that we thought would cause problems, where a lot of the messages has to go through a few nodes. As expected the queues of the nodes with a lot of traffic becomes large. VisuAAL gave a clear overview and it was easy to detect and visualize this behavior. We presented a visualization of AODVv2 protocol model to one of the authors of the model, who responded with the following:

> I find your visualisation very interesting, as it shows congestion in a realistic situation. One of the perhaps surprising outcomes I see is that congestion happens all over the place; it is less centered in the narrow corridor than one might have expected.
>
> *Rob van Glabbeek [17]*

In [29] we created models based on our understanding of Neocortec's MAC [27] and Routing [26] protocols. We want to stress that the model and our understanding of the protocol is not the same as Neocortec's protocols, since we do not have all the details. For the MAC protocol model, we visualized the slots that are chosen by nodes, and the discovery of neighbors. The MAC protocol utilizes synchronization and must synchronize to its neighbors to schedule a data transmission and is thus of high importance. We have modeled this as an output variable, which makes us able to visualize when data transmissions are scheduled. For the Routing protocol model, we show for a configuration with 2 sink nodes, how the race numbers progress in the network and how important it is to choose a maximum race number that is sufficiently large for the topology. For the topology in the routing experiments we discovered with a max race number of 5, the data could be routed in circles and not reach the sink node. When we

then increase the maximum race number to 10, a route to the sink node is taken. This clearly shows that a bad configuration of this could have large consequences.

Scalability of Protocol Models

Since we directly use the simulation feature of UPPAAL, we are dependent on time it takes to run simulations. Therefore we made a number of experiments concerning scalability of simulation time for protocol models in UPPAAL. We know from [29] that the Neocortec protocol models we created do not scale linearly with the number of nodes. In this project we tried a number of things to make simple models scale linearly with the number of nodes. We concluded that by sequentializing the actions performed by UPPAAL, the simulation time can be significantly decreased. We made two versions of a simple model of the Slotted ALOHA protocol. One version that follows the same approach as the models from our case studies and one where the flow was sequentialized. We then simulated both models with for instance 1000 nodes for 1000 model time units 10 times. The non-sequentialized model took 42922 seconds to simulate, while the sequentialized version took 86 seconds, which is almost a factor 500 difference. The approach does however set requirements to the model, which may not be suitable for large and complex models.

VisuAAL Usages

LinkAiders [33] and Neocortec [37] work in collaboration to create wireless devices, called Reachis, to be used during natural disasters in the Philippines. They intent to use thousands of devices, and are interested in simulating and learning more about the protocols and the physical locations of the devices. To support this, we have enabled VisuAAL to load a log file with location and connectivity data from their prototypes and use this topology to run it with their protocol. We did some initial experiments, to see how large log files we can handle with VisuAAL, but since the prototypes are not yet developed, we will not get to try it with the real log files. In order to load large UPPAAL models which contains many updates, we need to run the Linux version of UPPAAL which is 64-bit. On Linux we managed to load models with 86337 connectivity updates.

In [29] we created a MAC protocol model based on our understanding of Neocortec's MAC protocol. Ivanov [25] extended this model with energy consumption based on a kinetic energy model and created a wrapper for verifyta to output the data. We used this model and wrapper with VisuAAL to visualize the battery charge levels for the MAC protocol model, which gave insight in the energy consumption during the startup of the protocol. During the startup we can see that the energy use of each node seems to be dependent on the number of neighbors, such that many neighbors give a high energy consumption. This shows that we can use VisuAAL to track yet another type of data from a model, that gives useful insight in the model and protocol behavior.

We asked Ivanov for a comment for this report regarding VisuAAL, and the following is his response:

> Your visualization tool provides a comprehensive picture with flexible customization. It helps to assess the simulation results empirically and find weaknesses in the protocol and flaws in the model implementation. It's a huge aid for both protocol designers and common users.
>
> *Dmitry Ivanov [24]*

In conclusion, we have created a domain specific application capable of visualizing behavior of mesh network protocol models. VisuAAL is generic, such that any protocol in the domain can be visualized, as long as the model follows the few requirements. If the model is UPPAAL SMC compatible, it is rarely more than renaming of some variables that is needed before it is VisuAAL compatible. VisuAAL provides insight and overview of protocols, for different configurations and topologies, that would be cumbersome to get from actual devices.

12.2 Future Work

This section presents some interesting aspects which could be extended or improved upon. The tool is open-source and can thus easily be extended. We refer to Appendix E for a guide on how to get started on continuing our development.

12.2.1 Case Studies

Only a few people have used VisuAAL at this time. Feedback from multiple users could be very useful, both to improve the usability but also to get some pointers from users on what should be the focus for further development.

It could be interesting to have users using our tool to actually try to discover or explore a problem in their protocol, or maybe just use it while improving their protocol in different aspects.

12.2.2 Dynamic Topology in Topology Generator

Right now, it is only possible to generate a dynamic topology if one uses the GPS log functionality. However, a more user-friendly approach could be to add this functionality in the topology generator. Using a timeline, the user should be able to specify which changes to the topology that should occur at a specific time by dragging nodes around.

12.2.3 Automatic Error Detection

One of the ways that VisuAAL can help protocol developers, is when a developer is trying to find errors in his protocol. That can be done by writing a VQ which is satisfied when we reach some state which the user has specified as an error state, and then manually inspecting if the visualization marks any nodes or edges with this error state. It could be beneficial to add functionality which based on a VQ, could find the simulations and present the time in the

simulation where such occurrences took place, such that the user does not have to look through the simulations manually.

12.2.4 Additions to VisuAAL Query

Right now, we have two different types of VisuAAL Queries (VQ) when it comes to coloring. That could be greatly improved upon.

Instead of only allowing a gradient or some fixed colors, a combination could be added. An example could be to write a VQ that colors nodes with a gradient from red to blue, except if they are exactly one value that maps to the color green. Another example is to have multiple gradients, for instance from red to blue to yellow with multiple bounds.

In addition, GraphStream [22] allows many different ways to customize the graph. Instead of only changing the colors on edges or nodes as we do now, for instance the thickness of edges or the size of nodes could be changed. One could even add such that a node does not have to be round, but may be symbolized as a triangle. This could add great value to visualization of protocols like LMAC or the Routing protocol model inspired by Neocortec where there are different types of nodes.

Another addition to VQ could be to add `min` and `max` as reserved keywords which would map to the lowest or highest value for a given variable in a simulation across all nodes or edges. Currently, the bounds for VQs have to be set manually, meaning that the user has to either know these bounds beforehand, or manually inspect the simulation and find them before writing the VQ.

12.2.5 Increased Parsing Possibilities

In Section 4.3 on page 21 we described how the user can model locations as output variable. In UPPAAL it is possible to write queries, that check if we are in a given location, adding this directly in VisuAAL is possible. It might be useful if we were able to parse location names, perhaps with a prefix, and thus help the user to more easily examine this for models.

In Experiment 12.7 on page 99 we presented an improved model of the Slotted ALOHA protocol, which defines the topology differently than what we can parse right now. The improved version uses a sparse matrix. This type of topology representation is easy to parse, and it could be very beneficial in future protocol models.

12.2.6 User Interface

The user interface of VisuAAL is not as intuitive as we would like it to be. While this is an expert system where some amount of training is to be expected, there are certainly areas where the interactions should be better. It could be beneficial to let multiple users use VisuAAL, and analyze how the user interface could be improved.

There are essentially two kinds of features in VisuAAL. One is to generate a scenario the user wants to test. A scenario would include configuration, topology and dynamic updates. The other feature is to show results from (previous) simulations and investigate how the protocol model behaves, using VQs to describe the desired outputs. We have thought about making a more clear split of features in VisuAAL, so that these two types of functionalities become separate.

Additionally it would be beneficial to improve the result view with more sophisticated features like time control to be able to highlight details that require precise timing and zoom functionality which could help when analyzing larger topologies.

12.2.7 Run UPPAAL Remotely

Sometimes the simulations take a long time to complete. One way to reduce the simulation run time, could be to use a remote computer to run the simulations on. As an example, there is a super computer at AAU which could be beneficial to use when running simulations.

We have already worked a bit on this, by creating a web service which is hosted on the machine which executes the queries. When it receives a special web request in which the model and the query is included, it will start running verifyta, and return the answer when done. However, the part that uses this web service is not included in VisuAAL.

The source code for the tool, which runs UPPAAL remotely can be found here:
`https://github.com/lajtmaN/RemoteUPPAAL`

12.2.8 Background in Simulations

When the user generates a topology using the topology generator in VisuAAL, the user can relate the nodes' locations to a physical location shown on Google Maps. This is also the case when the user loads a GPS log file. However, the physical location is only visualized in the topology generator tab. To improve user experience, it could add great value if the Google Maps background was shown on the simulation results as well. We are unable to implement this at the moment, because of technical limitations of the Google Maps library we use. Due to time constraints we have not looked further into the solution to this, but a possible solution could be to use a static image based on the map from the topology generator.

13 Bibliographical Remarks

Because this report is written as the result of a multi-semester project, some parts have been reused from our report from last semester [29], and a few parts have been used multiple times in this report. The following lists the affected sections, and details the changes we have made to them in this report.

- The summary on page ii is based on text from other places in this report, including the abstract and the conclusion.
- Section 2.1 and Section 2.2 (excluding Section 2.2.1) are adapted versions from [29].
- Section 3.1 and Section 3.2 have been adapted from their respective uses in [29] for use in this report, with expanded examples and improved spelling.
- Section 8.1 and Section 9.1 are greatly shortened versions of the respective introductions to the MAC and Routing protocol by Neocortec presented in [29].

Also note that the models based on Neocortec's protocols were constructed in large during the last semester, with this semester being focused on extending, configuring and visualizing them.

Bibliography

[1] Ahirwar, Dilip Kumar, Verma, Prashant, and Daksh, Jitendra. “Enhanced AODV Routing Protocol for Wireless Sensor Network Based on ZigBee”. In: *Advances in Computer Science and Information Technology. Networks and Communications: Second International Conference, CCSIT 2012, Bangalore, India, January 2-4, 2012. Proceedings, Part I*. Ed. by Meghanathan, Natarajan, Chaki, Nabendu, and Nagamalai, Dhinaharan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 97–102. ISBN: 978-3-642-27299-8. DOI: 10.1007/978-3-642-27299-8_11. URL: http://dx.doi.org/10.1007/978-3-642-27299-8_11.

[2] Alnuaimi, M., Shuaib, K., and Jawhar, I. “Performance Evaluation of IEEE 802.15.4 Physical Layer Using MatLab/Simulink”. In: *2006 Innovations in Information Technology*. Nov. 2006, pp. 1–5. DOI: 10.1109/INNOVATIONS.2006.301905.

[3] Baran, P. “On Distributed Communications Networks”. In: *IEEE Transactions on Communications Systems* 12.1 (Mar. 1964), pp. 1–9. ISSN: 0096-1965. DOI: 10.1109/TCOM.1964.1088883.

[4] Behrmann, Gerd et al. “UPPAAL implementation secrets”. In: *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer. 2002, pp. 3–22.

[5] Bengtsson, Johan et al. “UPPAAL—a tool suite for automatic verification of real-time systems”. In: *Hybrid Systems III*. Springer, 1996, pp. 232–243.

[6] Bulychev, Peter E. et al. “UPPAAL-SMC: Statistical Model Checking for Priced Timed Automata”. In: *Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2012, Tallinn, Estonia, 31 March and 1 April 2012*. 2012, pp. 1–16. DOI: 10.4204/EPTCS.85.1. URL: http://dx.doi.org/10.4204/EPTCS.85.1.

[7] David, Alexandre et al. “Statistical Model Checking for Networks of Priced Timed Automata”. In: *Formal Modeling and Analysis of Timed Systems: 9th International Conference, FORMATS 2011, Aalborg, Denmark, September 21-23, 2011. Proceedings*. Ed. by Fahrenberg, Uli and Tripakis, Stavros. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 80–96. ISBN: 978-3-642-24310-3. DOI: 10.1007/978-3-642-24310-3_7. URL: http://dx.doi.org/10.1007/978-3-642-24310-3_7.

[8] David, Alexandre et al. “Timed I/O Automata: A Complete Specification Theory for Real-time Systems”. In: *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*. HSCC ’10. Stockholm, Sweden: ACM, 2010, pp. 91–100. ISBN: 978-1-60558-955-8. DOI: 10.1145/1755952.1755967. URL: http://doi.acm.org/10.1145/1755952.1755967.

[9] David, Alexandre et al. “Uppaal SMC tutorial”. In: *STTT* 17.4 (2015), pp. 397–415. DOI: 10.1007/s10009-014-0361-y. URL: http://dx.doi.org/10.1007/s10009-014-0361-y.

[10] Fehnker, Ansgar. *E-mail correspondence May 02, 2017*.

[11] Fehnker, Ansgar. *LMAC Protocol model*. URL: https://sites.google.com/site/fehnker/lmac.

[12] Fehnker, Ansgar, Hoesel, L. F. W. van, and Mader, A. H. “Modelling and Verification of the LMAC Protocol for Wireless Sensor Networks”. In: *Proceedings of the 6th International Conference on Integrated Formal Methods, IFM 2007, Oxford, Britain*. Ed. by J. Davis and J. Gibbons. Lecture Notes in Computer Science. Springer Verlag, July 2007, pp. 253–272.

[13] Fehnker, Ansgar et al. “Automated Analysis of AODV Using UPPAAL”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 – April 1, 2012. Proceedings*. Ed. by Flanagan, Cormac and König, Barbara. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 173–187. ISBN: 978-3-642-28756-5. DOI: 10.1007/978-3-642-28756-5_13. URL: http://dx.doi.org/10.1007/978-3-642-28756-5_13.

[14] Fehnker, Ansgar et al. “Topology-Based Mobility Models for Wireless Networks”. In: *Quantitative Evaluation of Systems: 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. Ed. by Joshi, Kaustubh et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 389–404. ISBN: 978-3-642-40196-1. DOI: 10.1007/978-3-642-40196-1_32. URL: http://dx.doi.org/10.1007/978-3-642-40196-1_32.

[15] Fluxicon. *Disco*. URL: http://fluxicon.com/disco.

[16] Gephi. *The Open Graph Viz Platform*. URL: http://gephi.org.

[17] Glabbeek, Rob van. *E-mail correspondence May 18, 2017*.

[18] GNU Organization. *GNU LESSER GENERAL PUBLIC LICENSE*. 2017. URL: https://www.gnu.org/licenses/lgpl-3.0.en.html.

[19] Google Inc. *Google Maps*. Accessed: May 2017. URL: https://maps.google.com.

[20] Gosling, James et al. *The Java® Language Specification*. 2017. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[21] GraphStream. *GraphStream - CSS reference*. 2017. URL: http://graphstream-project.org/doc/Advanced-Concepts/GraphStream-CSS-Reference/.

[22] GraphStream Team. *GraphStream - A Dynamic Graph Library*. 2010. URL: http://graphstream-project.org/.

[23] Höfner, Peter and McIver, Annabelle. "Statistical Model Checking of Wireless Mesh Routing Protocols". In: *NASA Formal Methods: 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*. Ed. by Brat, Guillaume, Rungta, Neha, and Venet, Arnaud. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 322–336. ISBN: 978-3-642-38088-4. DOI: 10.1007/978-3-642-38088-4_22. URL: http://dx.doi.org/10.1007/978-3-642-38088-4_22.

[24] Ivanov, Dmitry. *E-mail correspondence May 24, 2017*.

[25] Ivanov, Dmitry. "Kinetic Battery Model for Wireless Network Protocols with Power Consumption Scheme Approximated by Polynomials". Unpublished manuscript. 2017.

[26] Jaeger, Morten Gravild Bjerregaard. "Method and System for Routing Information in a Network". Patent 2012/0213227 A1 (US). Aug. 2012. URL: https://www.google.com.au/patents/US20120213227.

[27] Jaeger, Morten Gravild Bjerregaard. "Wireless Network Medium Access Control Protocol". Patent 2012/0120871 A1 (US). May 2012. URL: https://www.google.com/patents/US20120120871.

[28] Jeong, Dong Geun and Jeon, Wha Sook. "Performance of an exponential backoff scheme for slotted-ALOHA protocol in local wireless environment". In: *IEEE transactions on vehicular technology* 44.3 (1995), pp. 470–479.

[29] Jørgensen, Kevin H., Christoffersen, Niels B., Petersen, Rasmus D. and Gjøderum, Tim H. *Applying Statistical Model Checking to Wireless Mesh Network Protocols - A Case Study in Neocortec Protocols Using UPPAAL*. Jan. 2017.

[30] Larsen, Kim G, Pettersson, Paul, and Yi, Wang. "UPPAAL in a nutshell". In: *International Journal on Software Tools for Technology Transfer (STTT)* 1.1 (1997), pp. 134–152.

[31] Legay, Axel, Delahaye, Benoît, and Bensalem, Saddek. "Statistical model checking: An overview". In: *International Conference on Runtime Verification*. Springer. 2010, pp. 122–135.

[32] Lin, Fuchun J, Chu, PM, and Liu, Ming T. "Protocol verification using reachability analysis: the state space explosion problem and relief strategies". In: *ACM SIGCOMM Computer Communication Review*. Vol. 17. 5. ACM. 1987, pp. 126–135.

[33] LinkAiders. *LinkAiders*. [Accessed May 30, 2017]. URL: http://www.linkaiders.com/.

[34] Manwell, James F. and McGowan, Jon G. "Lead acid battery storage model for hybrid energy systems". In: *Solar Energy* 50.5 (1993), pp. 399–405. ISSN: 0038-092X. DOI: http://dx.doi.org/10.1016/0038-092X(93)90060-2. URL: http://www.sciencedirect.com/science/article/pii/0038092X93900602.

[35] MathWorks. *Simulink*. URL: https://mathworks.com/products/simulink.

[36] Neocortec. *Meeting at Neocortec March 02, 2017*.

[37] Neocortec. *neo.cortec*. [Accessed May 23, 2017]. URL: http://neocortec.com/.

[38] OMNeT++. *Discrete Event Simulator*. URL: https://omnetpp.org.

[39] Oracle. *JRE 8 Download*. 2017. URL: http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html#close.

[40] Parr, Terence. *ANTLR*. http://www.antlr.org/. Accessed: April 2017.

[41] Parr, Terence. *ANTLR 4 License*. 2017. URL: http://www.antlr.org/license.html.

[42] Perkins, Charles E. et al. *Ad Hoc On-demand Distance Vector Version 2 (AODVv2) Routing*. Internet-Draft draft-ietf-manet-aodvv2-16. Work in Progress. Internet Engineering Task Force, May 2016. 84 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-manet-aodvv2-16.

[43] Perkins, Charles E. et al. *Ad Hoc On-demand Distance Vector Version 2 (AODVv2) Routing*. Internet-Draft draft-perkins-manet-aodvv2-00. Work in Progress. Internet Engineering Task Force, Mar. 2017. 83 pp. URL: https://datatracker.ietf.org/doc/html/draft-perkins-manet-aodvv2-00.

[44] Perkins, Charles, Belding-Royer, Elizabeth, and Das, Samir. *Ad hoc on-demand distance vector (AODV) routing*. Tech. rep. 2003.

[45] Roberts, Lawrence G. "ALOHA Packet System with and Without Slots and Capture". In: *SIGCOMM Comput. Commun. Rev.* 5.2 (Apr. 1975), pp. 28–42. ISSN: 0146-4833. DOI: 10.1145/1024916.1024920. URL: http://doi.acm.org/10.1145/1024916.1024920.

[46] Sommer, C. and Dressler, F. "The DYMO Routing Protocol in VANET Scenarios". In: *2007 IEEE 66th Vehicular Technology Conference*. Sept. 2007, pp. 16–20. DOI: 10.1109/VETECF.2007.20.

[47] Texas Instruments. *CC1110 Overview Page*. [Accessed May 30, 2017]. URL: http://www.ti.com/product/CC1110-CC1111.

[48] The Apache Software Foundation. *Apache License*. 2017. URL: https://www.apache.org/licenses/LICENSE-2.0.

[49] The Eclipse Foundation. *Eclipse Public License - v 1.0*. 2017. URL: https://www.eclipse.org/legal/epl-v10.html.

[50] UPPAAL. *UPPAAL with Priorities*. June 2015. URL: http://www.it.uu.se/research/group/darts/uppaal/priority.shtml.

[51] Upsala University and Aalborg University. *UPPAAL Home Page*. 2017. URL: http://uppaal.org/.

[52] Van Hoesel, Lodewijk FW and Havinga, Paul JM. "A lightweight medium access protocol (LMAC) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches". In: (2004).

Appendices

A Updated UPPAAL Declaration CFG

We converted the CFG found in the UPPAAL help menu to ANTLR4 [40] g4 format, shown in the following listing.

```
grammar uppaal;
           xta : declaration* EOF;
   declaration : functionDecl | variableDecl | typeDecl | procDecl ;
 instantiation : ID '=' ID '(' argList ')' ';'
               | ID ':=' ID '(' argList ')' ';'
               ;

   systemBlock : instantiation* system ;
        system : 'system' ID (',' | '<') ID)* ';'  ;
 parameterList : '(' ( parameter ( ',' parameter )* )? ')' ;
     parameter : type ( '&' )? ID arrayDecl* ;
  functionDecl : type ID parameterList block ;
      procDecl : 'process' ID parameterList '{' procBody '}' ;
      procBody : (functionDecl | variableDecl | typeDecl)* states (commit)?
    ↪ (urgent)? init (transitions)? ;
        states : 'state' stateDecl (',' stateDecl)* ';' ;
     stateDecl : ID ( '{' expression '}' )? ;

        commit : 'commit' stateList ';' ;
        urgent : 'urgent' stateList ';' ;
     stateList : ID (',' ID)* ;
          init : 'init' ID ';' ;
   transitions : 'trans' transition (',' transitionOpt)* ';'  ;
    transition : ID '-' ID transitionBody ;
 transitionOpt : transition | '-' ID transitionBody  ;
transitionBody : '{' (guard)? (sync)? (assign)? '}' ;
         guard : 'guard' expression ';' ;
          sync : 'sync' expression ('!' | '?') ';' ;
        assign : 'assign' exprList ';' ;
      typeDecl : 'typedef' type typeIdList (',' typeIdList)* ';'  ;
    typeIdList : ID arrayDecl* ;
  variableDecl : type declId (',' declId)* ';'  ;
        declId : ID arrayDecl* ( '=' initialiser )?
               | ID arrayDecl* ( ':=' initialiser )? ;

   initialiser : expression
               |  '{' fieldInit ( ',' fieldInit )* '}'
               ;
     fieldInit : ( ID ':' )? initialiser ;

     arrayDecl : '[' expression ']' ;

          type : prefix ID ( range )?
               |  prefix 'struct' '{' fieldDecl+ '}'
               ;
```

```
45
46      fieldDecl : type fieldDeclId (',' fieldDeclId)* ';' ;
47    fieldDeclId : ID arrayDecl* ;
48
49         prefix : ( ( 'urgent' )? ( 'broadcast' )? | ('const' )? | ('meta')? ) ;
50          range : '[' expression ',' expression ']' ;
51
52          block : '{' ( variableDecl | typeDecl )* statement* '}' ;
53
54      statement : block
55                | ';'
56                | expression ';'
57                | 'for' '(' exprList ';' exprList ';' exprList ')' statement
58                | 'for' '(' ID ':' type ')' statement
59                | 'while' '(' exprList ')' statement
60                | 'do' statement 'while' '(' exprList ')' ';'
61                | 'if' '(' exprList ')' statement ( 'else' statement )?
62                | 'break' ';'
63                | 'continue' ';'
64                | 'switch' '(' exprList ')' '{' caseExpr+ '}'
65                | 'return' ';'
66                | 'return' expression ';'
67                ;
68
69     caseExpr   : 'case' expression ':' statement*
70                |  'default' ':' statement*
71                ;
72
73       exprList : expression ( ',' expression )* ;
74     expression : ID
75                | NAT
76                | FLOAT
77                | 'true'
78                | 'false'
79                | ID '(' argList ')'
80                | expression '[' expression ']'
81                | '(' expression ')'
82                | expression '++'
83                | '++' expression
84                | expression '--'
85                | '--' expression
86                | expression assignOp expression
87                | unaryOp expression
88                | expression rel expression
89                | expression binIntOp expression
90                | expression binBoolOp expression
91                | expression '?' expression ':' expression
92                | expression '.' ID
93                | expression '\''
94                | 'exists' '('ID ':' type ')' expression
95                | 'forall' '('ID ':' type ')' expression
96                ;
```

```
97
98        argList : (expression ( ',' expression )* )? ;
99
100      assignOp : '=' | ':=' | '+=' | '-=' | '*=' | '/=' | '%='
101               | '|=' | '&=' | '^=' | '<<=' | '>>=' ;
102       unaryOp : '-' | '!'  | '+' ;
103           rel : '<' | '<=' | '==' | '!=' | '>=' | '>' ;
104      binIntOp : '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '<<' | '>>' ;
105     binBoolOp : '&&' | '||' ;
106
107            ID : [a-zA-Z_]([a-zA-Z0-9_])* ;
108           NAT : [0-9]+ ;
109         FLOAT : [0-9]+('.'[0-9]+)? ;
110            WS : [ \n\t\r]+ -> channel(HIDDEN) ;
111  BLOCK_COMMENT : '/*' .*? '*/' -> channel(HIDDEN) ;
112   LINE_COMMENT : '//' ~[\r\n]* -> channel(HIDDEN) ;
```

B AODVv2 Model Connectivity Function Change

```
bool isconnected(IP i, IP j) {
  return(topology[i][j]==1);
}
```

Listing B.1: Original isconnected function

```
bool isconnected(IP i, IP j) {
  if(i == 0 || j == 0) {
    return false;
  }
    return(CONFIG_connected[i-1][j-1]==1);
}
```

Listing B.2: Updated isconnected function

C ANTLR4 VQ Syntax

Here we present the full syntax for the VQ language in ANTLR4 [40] g4 format, for help in understanding the language and especially the parser for further development. The precedence of expressions is higher for the topmost rules compared to the lower rules, such that for instance $*$ has higher precedence than $+$.

```
grammar vq;
query
      : (gradient | colors)? exp EOF;

gradient
      : '[' oneGradient ',' oneGradient ']';

oneGradient
      : ID ':' NEG? FLOAT
      | ID;

colors
      : '[' color+ ID ':' '*' ']';

color
      : ID ':' NEG? FLOAT ',';

exp
      : '(' exp ')'                                   #par
      | <assoc=right> op ='-' exp                     #unOp
      | <assoc=right> op ='!' exp                     #unOp
      | exp op=('*' | '/') exp                        #binOp
      | exp op=('+' | '-') exp                        #binOp
      | exp op=('<' | '<=' | '>' | '>=') exp          #binOp
      | exp op=('==' | '!=') exp                      #binOp
      | exp op='&&' exp                               #binOp
      | exp op='||' exp                               #binOp
      | <assoc=right> exp '?' exp ':' exp             #condOp
      | ID                                            #id
      | ID '.' ID                                     #idDot
      | FLOAT                                         #float
      | BOOL                                          #bool
      ;

BOOL  : 'true' | 'false';
NEG   : '-';
ID    : [a-zA-Z_]([a-zA-Z0-9_])*;
FLOAT : [0-9]+('.'[0-9]+)?;
WS    : [ \n\t\r]+ -> channel(HIDDEN);
```

D GPS Log Example

The following listing contains the first 50 entries of the 50100 entries long GPS log file to generate the dynamic topology shown in Section 10.1.1. The entire GPS log file, can be found on Appendix ZIP and is named `GPSLog_dynamic_0_1.txt`.

```
0;0;57.010966;9.968164;45 21 49 5 76 68 83 21 94 63
1000;0;57.010983408;9.96817346156909;45 22 49 5 76 68 83 21 94 64
2000;0;57.0110637654277;9.96828975499679;45 26 49 10 76 73 83 24 94 67
3000;0;57.0110128812554;9.96827480682454;45 25 49 8 76 71 83 22 94 65
4000;0;57.0109336375218;9.96842480255822;45 29 49 10 76 70 83 19 94 63
5000;0;57.0110335370602;9.96847814655822;45 32 49 14 76 74 83 23 94 66
6000;0;57.0110509450602;9.96861071505864;45 36 49 17 76 76 83 22 94 66
7000;0;57.0109843770559;9.96866405905864;45 37 49 17 76 74 83 20 94 63
8000;0;57.0110017850559;9.96880736394921;45 41 49 20 76 74 83 19 94 62
9000;0;57.0109531848477;9.96886070794921;45 42 49 20 76 72 83 17 94 60
10000;0;57.0110461603577;9.96891405194921;45 45 49 23 76 75 83 20 94 62
11000;0;57.0111513494315;9.96905517702299;45 51 49 29 76 76 83 22 94 63
12000;0;57.011224790017;9.96905248843746;45 52 49 31 76 78 83 24 94 65
13000;0;57.0111440538503;9.96910583243746;45 52 49 30 76 75 83 21 94 61
14000;0;57.0112095267797;9.96920724136686;45 56 49 33 76 75 83 21 94 61
15000;0;57.0111636897951;9.96919734038224;45 55 49 32 76 74 83 20 94 60
16000;0;57.0110894051442;9.96934237703312;45 58 49 32 76 69 83 16 94 55
17000;0;57.0111053662501;9.96939716792718;45 60 49 34 76 68 83 15 94 54
18000;0;57.0111767892591;9.96945051192718;45 63 49 36 76 68 83 17 94 54
19000;0;57.0112116889811;9.96952134764924;45 66 49 39 76 67 83 16 94 53
20000;0;57.0112290969811;9.96963806486359;45 69 49 41 76 64 83 15 94 50
21000;0;57.0112422353113;9.96968713919374;45 71 49 42 76 63 83 14 94 49
22000;0;57.0112779083556;9.96972221814943;45 72 49 43 76 62 83 14 94 48
23000;0;57.0113536780977;9.96971720040735;45 74 49 45 76 63 83 17 94 50
24000;0;57.0113710860977;9.96975370401628;45 75 49 46 76 62 83 16 94 49
25000;0;57.0114295042506;9.96976603786337;25 1 45 76 49 48 76 63 83 17 94 49
26000;0;57.011390736796;9.96987555731791;25 1 45 79 49 48 76 59 83 14 94 46
27000;0;57.0114935872209;9.96992890131791;25 5 45 82 49 52 76 58 83 16 94 45
28000;0;57.0114235000028;9.97006974053603;25 5 45 84 49 50 76 53 83 11 94 40
29000;0;57.0114409080028;9.9701419653349;25 6 45 87 49 51 76 51 83 9 94 38
30000;0;57.0114583160028;9.97012313362154;25 7 45 87 49 52 76 51 83 10 94 39
31000;0;57.0114544625521;9.97017647762154;25 7 45 88 49 52 76 50 83 9 94 37
32000;0;57.0114951979291;9.97025314899851;25 9 45 90 49 53 76 47 83 8 94 35
33000;0;57.0115126059291;9.97033564465536;25 11 45 92 49 54 76 45 83 6 94 32
34000;0;57.0115300139291;9.97046218900166;25 12 45 92 49 54 76 41 83 3 94 28
35000;0;57.0116236098406;9.97043934509013;25 15 45 95 49 57 76 42 83 6 94 30
36000;0;57.0116177535095;9.97049268909013;25 15 45 94 49 57 76 40 83 4 94 28
37000;0;57.0116471677799;9.97054603309013;25 17 45 94 49 57 76 38 83 3 94 26
38000;0;57.0116645757799;9.97052581293689;25 17 45 95 49 58 76 39 83 4 94 27
39000;0;57.0115926724475;9.9706684682693;25 16 45 89 49 55 65 2 76 34 94 22
40000;0;57.0116945288592;9.97080626068103;25 20 45 86 49 56 65 7 76 30 94 18
41000;0;57.0117011929393;9.97087034860099;25 20 45 84 49 56 65 10 76 28 94 16
42000;0;57.01171556303;9.9709206546917;24 0 25 21 45 82 49 55 65 11 76 26 94 15
43000;0;57.0116341970306;9.9709739986917;24 1 25 18 45 80 49 52 65 12 76 24 94 12
44000;0;57.0116516050306;9.97100242173092;24 2 25 19 45 80 49 52 65 14 76 23 94 12
45000;0;57.0115752392032;9.97114953955828;24 6 25 16 37 0 45 74 49 47 65 18 76 18 94 6
46000;0;57.0115156574099;9.97127987335163;24 9 25 14 37 4 43 0 45 70 49 43 65 21 76 14 94 2
47000;0;57.0115330654098;9.97128815615444;24 10 25 15 37 5 43 1 45 69 49 43 65 22 76 13 94 1
48000;0;57.0114671751742;9.97142479839006;24 13 25 12 37 9 43 4 45 64 49 39 65 25 76 9
49000;0;57.0114803194071;9.97147387862291;24 15 25 12 37 11 43 6 45 63 49 38 65 27 76 7
50000;0;57.0114974788712;9.971526974087;24 17 25 12 37 13 43 8 45 61 49 37 65 29 76 5
```

Listing D.1: GPS Log Example

E Use & Installation Guide for VisuAAL

The flowchart in Figure E.1 sketches how we intend VisuAAL to be used for loading models and simulations.

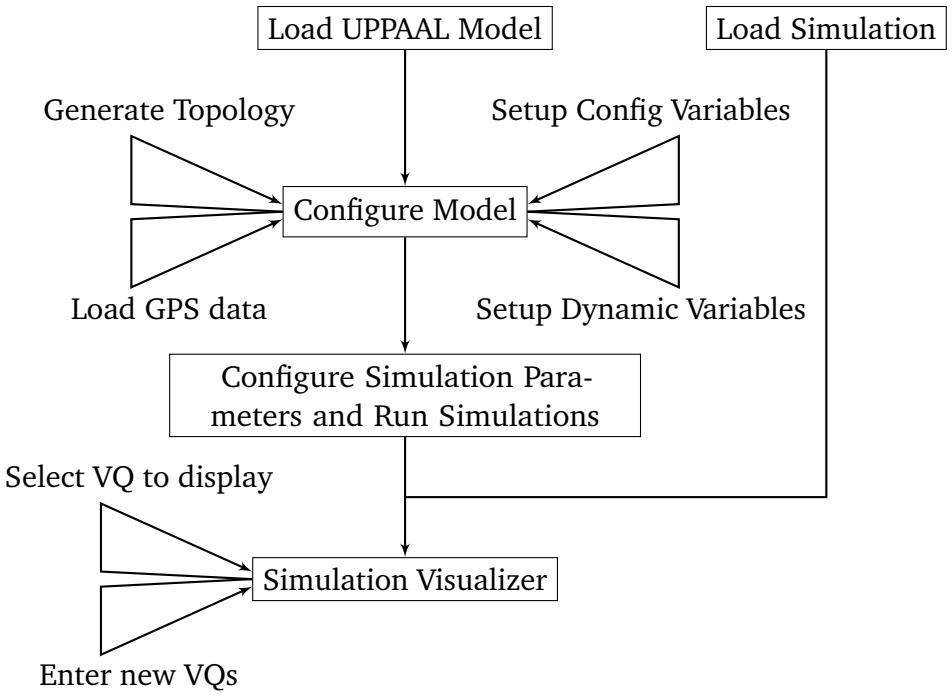


Figure E.1: Flow chart for intended application use

E.1 Running VisuAAL

All members in our project groups have been using computers running Windows 10 as operating system during development and in addition we have used a Linux server to run VisuAAL. Thus we have only tested VisuAAL for those operating systems. We do not guarantee that it works as expected on other platforms. To run VisuAAL install and download the following:

JRE 8 As VisuAAL is written in Java SE 8 [20], it is required to have Java SE Runtime Environment 8 [39] (JRE 8) installed. When JRE 8 is installed one can simply execute the `VisuAAL.jar` which starts VisuAAL.

UPPAAL - Verifyta To do simulations it is required to have the verifyta executable installed, that comes with UPPAAL [51]. VisuAAL will prompt you when it needs to know where it is located.

Download Application The latest version can be downloaded at: `https://github.com/lajtmaN/VisuAAL/releases`

Following is a guide to using VisuAAL.

1. Download and install UPPAAL (we have used 4.1.19 exclusively): http://uppaal.org/
2. Download and install Java SE Runtime Environtment (JRE) (we have used Java 8 update 121 exclusively): http://www.oracle.com/technetwork/java/javase/downloads/server-jre8-downloads-2133154.html
3. Download the newest version of VisuAAL: https://github.com/lajtmaN/VisuAAL/releases
4. Extract the downloaded zip folder
5. Run `VisuAAL.jar` from the extracted zip folder
6. Click `Load Model or Simulation`
7. Find your model, if you do not have one yourself you can load either DYMO or LMAC, located in the UPPAAL Models folder
8. After a model is loaded it can be configured, a topology can be generated and the updated model can be saved by pressing the `Save Model` button in the top right
9. When the desired configurations are made, a simulation can be run in the simulation tab, by specifying model `Time Bound`, `Number Of Simulations`, which topology to use (generated or topology specified in the model) and optionally a name for the simulation
10. If the verifyta path has not yet been chosen, VisuAAL will prompt and ask for the path. It is found in the installation folder of UPPAAL/bin-Win32 or UPPAAL/bin-Linux.
11. UPAALL will run, and when finished the result will be opened in a new tab. At the same time, the result will be saved in a folder called `simulations`, where you started the program. The new simulation will have the name set previously, or as default be named `Result`. Any previous file with the same name will be overwritten.
12. In the Result tab one can write queries in `Display` and these can be saved using the Export VQ's functionality. In the VQ Help, you can also see which variables that can be used for both edges and nodes.
13. To load a previous simulation, press `Load Model or Simulation` and change the file type in the window to a simulation file, and locate the file you want to load.

E.2 Compilation and Further Development

In order to compile VisuAAL for further development, we recommend the following tools and sources:

IDE IntelliJ IDEA Integrated Development Environment: https://www.jetbrains.com/idea/

JDK 8 Java Development Kit 8: http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html

Dependencies It is important that dependencies are handled in the IntelliJ IDEA for VisuAAL to be compiled. The dependencies are included in the IntelliJ project file.

VisuAAL Source Code `https://github.com/lajtmaN/VisuAAL`

VisuAAL Javadoc `https://lajtman.github.io/VisuAAL`

F Licensing

Since VisuAAL is dependent on a number of libraries, VisuAAL is also under the license restrictions of these libraries.

Below is the list of libraries we use and what license they are under.

ANTLR4 BSD 3-clause License [41]

GraphStream GNU Lesser General Public License v3 [18]

JUnit Eclipse Public License 1.0 [49]

GMapsFX Apache License, Version 2.0 [48]

Neither 4 of these licenses impose that we release our software under a special license, and for this reason we are free to choose a license as long as it does not conflict with our needs. We choose to use *Apache License, Version 2.0* [48] for VisuAAL, because it is one of the licenses that we are dependent on, and because it seems to fulfill our needs.

ZIP Contents of Attached ZIP

Along with the report a zip folder is included with the following contents:

- /
  - report.pdf ..... This report
  - VisuAAL/
    - VisuAAL GitHub ..... Link to GitHub repository
    - binaries/
      - VisuAAL.jar ..... The executable for VisuAAL
      - lib/
  - LMAC/
    - lmac513_SMCmodified.xml ..... Model for the LMAC protocol
    - simulations/ ..... Contains the simulations for the LMAC protocol
  - AODVv2/
    - dymo_SMCmodified.xml ..... Model for the AODVv2 protocol
    - simulations/ ..... Contains the simulations for the AODVv2 protocol
  - LinkAiders/
    - topology_only.xml ..... Model with nothing else than topology
    - GPSLog_dynamic_0_1.txt ..... GPS Log file example
    - simulations/ ..... Simulations with dynamic topology

For the printed versions of the report, additional confidential files are included. The file structure for this version is as follows:

- /
  - ..... All the previously mentioned files are also included
  - MAC_energy/ ..... Contains a simulation with energy
  - MAC/
    - mac_model.xml ..... Model for the MAC protocol
    - simulations/ ..... Contains the simulations for the MAC protocol
  - Routing/
    - routing_model.xml ..... Model for the Routing protocol
    - routing_model_revised.xml ..... Model for the revised Routing protocol
    - simulations/ ..... Contains the simulations for the Routing protocol