TU WIEN Informatics

Evaluating the Energy Simulations of Multicast Routing Protocols used in Wireless Sensor Networks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Daniel Lukitsch, BSc
Matrikelnummer 01634053

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Univ.Ass. Dipl.-Ing. Philipp Raich, BSc

Wien, 6. Dezember 2022

Daniel Lukitsch	Wolfgang Kastner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Evaluating the Energy Simulations of Multicast Routing Protocols used in Wireless Sensor Networks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Daniel Lukitsch, BSc
Registration Number 01634053

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Univ.Ass. Dipl.-Ing. Philipp Raich, BSc

Vienna, 6th December, 2022

Daniel Lukitsch	Wolfgang Kastner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Daniel Lukitsch, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. Dezember 2022

Daniel Lukitsch

Danksagung

An dieser Stelle möchte ich allen danken, die mich bisher während meines Master-Studiums und der Arbeit an dieser Diplomarbeit unterstützt haben.

Besonderen Dank möchte ich dabei Dipl.-Ing. Philipp Raich aussprechen, der sich regelmäßig Zeit nahm, um mich bei der Erstellung dieser Arbeit immer wieder durch neue Ideen, Hinweise zu unterstützen.

Weiters möchte ich auch Prof. Wolfgang Kastner für die Abwicklung dieser Arbeit, das interessante Thema und die ausführliche Begutachtung danken.

Abschließend möchte ich mich natürlich auch bei meiner Verlobten, meinen Eltern und allen, die mich sonst noch während des Master-Studiums unterstützt haben bedanken, denn ohne diese Unterstützung wäre all das nicht möglich gewesen.

Acknowledgements

I want to thank everyone who supported me with my master's studies and especially during the work of this master thesis.

Especially Dipl.-Ing. Philipp Raich, for the ideas, help and overall support he has given to me, and for the time he spent helping me with this thesis. Other thanks go to Prof. Wolfgang Kastner for making this exciting work possible and correcting it.

Finally, I want to thank my fiancée, parents, and everyone else who made it possible to finish my master's degree.

Kurzfassung

Durch die rasante Entwicklung und Verbreitung des *Internets der Dinge (IoT)* und des damit verbundenen, drastisch erhöhten Vernetzungsgrades von modernen Elektrogeräten stellt diese aktuelle Veränderung auch die etablierte Domäne der *drahtlosen Sensornetzwerke (WSNs)* vor neue Herausforderungen. Durch die Kombination des *Internet Protokolls Version 6 (IPv6)* mit dem Übertragungsstandard *IEEE 802.15.4* ist es nun möglich, diese ressourcen-limitierten Netzwerke in das *Internets der Dinge* einzubinden und dadurch neue praktische Anwendungen umzusetzen.

Diese neuen Möglichkeiten eröffnen jedoch auch weitere Problemstellungen, vor allem für Routing in diesen drahtlosen Sensornetzwerken. Die hierfür verwendeten Routingprotokolle müssen für eine möglichst verlässliche Ende-zu-Ende Übertragung jedes einzelnen Datenpakets sorgen, während die Übertragungen zusätzlich mit maximaler Energieeffizienz erledigt werden müssen. Um diese Eigenschaften für ein solches drahtloses Sensornetzwerk mit spezieller Topologie und Konfiguration zu ermitteln, sind moderne Netzwerksimulatoren unerlässlich.

Aus diesem Grund werden verschiedene Netzwerksimulatoren mit besonderem Fokus auf die Verwendung mit drahtlosen Sensornetzwerken evaluiert. Nach dieser Evaluierung der verfügbaren Simulatoren wird der *Network Simulator 3 (NS-3)* für die weitere Arbeit verwendet, um das *Multicast Protocol for Low-Power and Lossy Networks (MPL)* zu implementieren, das bestehende *Ad hoc On-Demand Distance Vector (AODV)*-Modell für die Verwendung mit IPv6 portiert und die bestehende Implementierung des *IEEE 802.15.4* Modells um ein Energiemodell zur direkten Simulation des Energieverbrauchs erweitert.

Basierend auf diesen Implementierungen wird der *NS-3* Simulator benutzt, um *Mesh-Under Flooding*, *MPL* und *AODV6* mittels verschiedenster Parameter-Konfigurationen und Netzwerktopologien vergleichen zu können. Diese Vergleiche erlauben es, die unterschiedlichen Konfigurationsmöglichkeiten der genannten Protokolle hinsichtlich ihrer Energieeffizienz, Verlässlichkeit und Störanfälligkeit zu beurteilen und aus diesen Simulationsergebnissen, Empfehlungen für die Konfiguration und Verwendung dieser Protokolle in realen Applikationen abzuleiten.

Abstract

In the past few years, the tremendous increase of connected and interlinked electronic devices led to the creation of the so-called Internet of Things (IoT), which has had a high impact on today's electronic devices, especially in the domain of Wireless Sensor Networks (WSNs). The development of 6LoWPAN, which enables WSN for the usage of IPv6 together with the transmission standard *IEEE 802.15.4* opens many possibilities for modern WSNs. The combination of these two technologies enables the direct connection of WSNs into the global IoT, and therefore the realization of newer and even bigger applications than it would have been possible with conventional and only locally connected WSNs.

These emerging possibilities create new use cases and lead to new challenges and requirements for modern WSNs. One of these challenges is the usage of specialized routing protocols inside the resource-limited nodes of a WSN. These specialized routing mechanisms have to maintain a stable end-to-end communication over an unstable physical transmission channel while at the same time only consuming an absolute minimum of energy to enable a maximum lifetime for a node with its given battery power. Accordingly, state-of-the-art network simulators are essential tools, as they allow us to simulate diverse WSN, their different network topologies and protocol configurations.

Therefore, we evaluate different network simulators for their usage and simulation of WSNs. As a result of this simulator evaluation, we further use the NS-3 and implement the Multicast Protocol for Low-Power and Lossy Networks (MPL) for this simulator. Additionally, we port the existing IPv4 based model for the Ad hoc On-Demand Distance Vector (AODV) routing protocol for the usage with IPv6. Further, we extend the simulator's *IEEE 802.15.4* model with an energy model, which allows us to simulate a node's energy consumption and turn it off if its battery uses all its available energy.

Further we use our extended NS-3 simulator to model the behaviour of WSNs that use the following routing protocols, MPL, *Mesh-Under Flooding* and Ad hoc On-Demand Distance Vector (AODV) Routing for IP version 6 (AODV6) with different parameter configurations on the most common network topologies. We then use the data and results generated in these simulations to compare the protocols against each other regarding energy efficiency, reliability and capabilities. Therefore, we want to provide developers of WSNs with recommendations and hints on how to use and configure these routing protocols to achieve the best performance and energy efficiency.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Modern WSNs have greatly improved and evolved in recent years. With the current hardware and software technology available, their common use-cases have changed. They are no longer separated networks which are designed for a single purpose. WSNs previously needed particular transmission and routing protocols, such that they were not compatible with the ones used in our standard computers. Instead, with the improvements and possibilities that IPv6 offers, we can even integrate whole WSNs into the global network of connected and intelligent devices, commonly referred to as the IoT [UWM+15]. Furthermore, this extends the usage of WSNs to many more applications, like smart agriculture, home and building automation, weather measurement systems and many others. However, all these applications have one thing in common: they all try to optimize and connect single data points into one connected system.

Therefore, we have to take lots of different factors into account. One of the most important ones is that the underlying WSN stays operational for a maximum time. Since nodes in a WSN are commonly battery-powered, the energy efficiency of every single node is critical in keeping the whole network operational. Hence, it is essential to simulate and test the energy efficiency within virtual networks and implement state-of-the-art routing protocols to keep the available network simulators up to date. The knowledge obtained from the simulation results of these virtual WSNs, then helps us to gain a better understanding of networks behaviour and its utilized network stack. This additional information can then be further used to configure the protocols within the utilized network stack in a more energy efficient way, for both virtual and also real-world wireless sensor nodes. This may then reduce the overall energy consumption which leads to a longer battery lifetime and ensures that the WSN stays operational for a longer period of time.

To reach these important goals, this thesis aim lies on the simulation based evaluation of multicast routing protocols for WSN. Due to the small and constrained amount of

energy that is available to wireless sensor nodes, the focus is especially on the energy efficiency, reliability and functionality of the utilized routing protocols. Therefore, we will have to evaluate and compare different wireless network simulators to select the best suiting one four our needs and identify missing features and simulation models, that we will have to implement ourselves. Next, after the simulator supports all the necessary functionalities we perform various simulations of WSNs with different network topologies and routing protocol configurations. The results obtained from these simulations are then compared for their energy efficiency and overall functionality, which provides us with a better understanding of the nodes' observed behaviour. This additional knowledge may then be used to find an optimal routing protocol configuration for a certain WSN and extend its overall battery lifetime in and outside of the virtual simulation setup.

1.2 Problem Statement

This thesis addresses multiple problem statements and focuses on multicast transmissions within the domain of WSN and their respective routing protocols. Due to the typically very limited and minimal amount of energy that is available to the wireless nodes of a WSN, it is a absolutely crucial to minimize the nodes' energy consumption to reduce the amount of necessary maintenance to ensure a maximal lifetime of the whole network.

This already introduces another problem that occurs, while implementing and fine trimming the behaviour of a WSNs. This central problem is the sheer size of the network itself, which often requires a lot of hardware nodes, a minimum distance between the nodes and a realistic environment to optimally configure a network for its desired use case. Most times creating, maintaining and rearranging such network setups solely for a small amount of test cases is completely impractical and out of scope.

In such a situation, it is inevitable to simulate the desired WSN within a virtual setup by using a special simulator for WSNs and evaluate the obtained results for this virtual network. As there are numerous different WSN simulators available, another problem is to find the one that suits best.

Therefore, we evaluate several available network simulators in chapter 2, to find a scalable and feature-rich simulator. Further, this simulator shall also be convenient to use and allows us to contribute our models and perform state-of-the-art simulations to analyse the energy efficiency and other properties of multicast transmissions in modern WSNs. We primarily focus on multicast routing protocols as they are commonly used with WSNs and provide a good foundation for further work.

This focus on multicast routing protocols comes form the typical use case of the wireless sensor nodes within a WSN. As the name suggests, these nodes are often smart and stand alone sensors that periodically transmit their measured values to every other receiver node within its transmission range in a reliable way, while consuming a the least amount of energy. For such a behaviour, multicast transmissions are the only feasible choice, when implementing a one-to-many communication scheme.

The importance of multicast transmissions raises a further problem on how these specific routing protocols solve their routing tasks while handling the inherent problem of achieving an energy-efficient and simultaneously reliable way to communicate in low power and lossy networks. To answer this question, we evaluate and analyse the functionality of different multicast routing protocols and how their performance changes in various different network and protocol configuration setups.

The second part of this work tries to answer what possibilities a simulator has to measure the energy consumption and efficiency of a wireless sensor node in a WSN and what metrics we can implement to quantify the energy consumption. This step is necessary as we are working in a completely virtual setting without the ability to physically measure the energy consumption of a real hardware node. Instead, we have to use energy models that try to mimic the real behaviour of a hardware node in the virtual simulation domain and provide us with approximated results for a node's energy consumption in the real world. Therefore, we examine several typical use cases together with specific routing protocol configurations to evaluate their impact on the nodes' energy efficiency, usability and reliability. These use cases consider the nodes within the WSN as static and provide the nodes with energy sources to mimic a real network with nodes that deplete their batteries and become dysfunctional. We then use these obtained simulation results to draw conclusions for a networks real-world behaviour and how realistic the behaviour within a simulator actually is.

Beneath surveying the accuracy and limitations of such network simulations, we want to evaluate metrics for different network topologies, protocol configurations and interference situations. Further, we want to derive limitations and recommendations for real-world applications from our results obtained from various use-case simulations. Based on our virtual simulation results, these conclusions will help us maximize real-world applications' performance and of used multicast routing protocols.

From all these different aspects and aims of this work, we derived the following four main research questions that we want to answer with the work conducted throughout this thesis:

- **RQ1: How can we measure a nodes' energy consumption in the virtual environment of a simulator?**
  What possibilities are there to determine the realistic energy consumption of a virtually simulated node within a WSN.
- **RQ2: How accurate are virtual simulation results compared to physical measurements?**
  What are the limitations and general restrictions of the utilized energy consumption models, how can we improve their accuracy, and what does this mean for their overall applicability.
- **RQ3: Which hardware-independent metrics can we apply to acquire a realistic overview of a WSNs energy consumption?**

Are some metrics not connected to any hardware-specific properties, which we can use to get a reliable approximation of the energy efficiency of a routing protocol for WSNs.

- **RQ4: How can we define the energy-efficiency of a routing protocol?**
  Which network and transmission parameters of a node can we use as a metric to compare the energy efficiency of a routing protocol.

1.3 State of the Art

WSN nodes and their running software must be permanently optimized to maximize the performance and energy efficiency of the whole network they form. Peculiarly as the nodes' hardware and software capabilities improve at a very high rate, Lahmar et al. show the improvements for modern nodes' processors [LCA12], which provide us with a boost in processor speed and the increase of available memory. Even more important is the improvement of the overall energy efficiency of modern hardware, especially on the energy needed to send and receive packets, as is shown by Adelmann in [Ade21]. The physical measurements done by Adelmann show that for modern hardware, the energy used to receive and transmit has dropped to similar levels as the actual energy consumption of the used processor itself, which makes sending and receiving data less critical for the overall energy consumption than it was with older hardware.

Another important milestone for a WSN's functionality is the definition of the *IEEE 802.15.04* physical layer standardization [IEE20]. Together with the definition of the 6LoWPAN network adaption layer [MHCK07], these two protocols are essential for the interoperability and connectivity between nodes from different WSN and their overall integration into the IoT.

These innovations in the nodes network stack led to the need of a broad range of new energy efficient unicast and multicast routing protocols that can be efficiently used within WSNs. According to Raich et al. these routing protocols can be classified by their utilized approach to route the messages [RK18]. These most common protocol classifications are:

- **Flooding-Based**
  These protocols utilize the intrinsic broadcasting behaviour of wireless transmissions. An initial sending node sends its data message. Depending on the utilized protocol algorithm, all the receiving nodes either retransmit a message to their neighbours or ignore it. The most famous protocols from this class are the basic *Flooding* protocol and to some extinct also the MPL routing protocol, that uses the classical flooding approach combined with a probabilistic algorithm to route its data messages.
- **Table-Driven (Proactive)**
  For these protocols, the nodes try to create and maintain a representative graph of the current network within their memory (tables) to identify the optimal route to a message that must be sent. These protocols require a permanent and periodic

update and check if their maintained routing tables perform efficiently and register changes within the network. Typical protocols for this protocol class are the Routing Protocol for Low power and Lossy Networks (RPL) and Low-Energy Adaptive Clustering Hierarchy (LEACH) routing protocols.

- **Ad-Hoc (Reactive)**
  The *ad-hoc* or *reactive* routing protocols utilize the completely opposite approach as the previously introduced *table-driven* ones. These protocols do not maintain a graph that represents the current network topology. Instead, they always try to acquire the necessary routing information *on-demand* when they want to send a new data message. Therefore, before sending the actual data message, reactive protocols perform some sort of *route request*, where they try to determine the network route to send the datagram to. An example of such an *reactive* protocol is the AODV protocol, which was initially developed for use within Mobile Ad-hoc Networks (MANETs). Due to continuous improvements, it is also used within WSNs.

For completion, there are also the *multipath*, *hybrid* and *optimization* based approaches that may offer better performance than the protocols from the previously introduced protocol classes. However, these protocols also have a very high computational complexity, resulting in high energy consumption, which reduces a node's lifetime, which is often not feasible for a WSN.

Another approach to classify and evaluate the numerous available routing protocols is made by Sobral et al. [SRR+19]. This work analyses several routing protocols for their use within WSNs and classifies them by their functional capabilities like their ability to perform multicast transmissions, their capability to support mobile nodes and their achieved Quality of Service (QoS). For this work, we are mainly focused on the listed multicast protocols like the flooding-based MPL [HK16], which is specially designed to transmit multicast messages within a low power and lossy network environment like a WSN. Additionally, Sobral et al. also analyse and compare the different enhancements for the basic RPL routing algorithm [SRR+19] [WTB+12], which make this solely unicast-based and *table-driven* routing protocol capable for the transmission of multicast messages within a WSN.

However, aside from the physical hardware and its increase in performance and resources over the last years, the software running on a modern WSNs node is as important and essential as the hardware it runs on. The software has to utilize its available hardware resources optimally and handle the intrinsic restrictions and problems of WSNs as discussed in [PK18]. Although some of these problems, achieving a *low transmission delay*, were already present in the domain of *ad-hoc networks*, the developed solutions for *ad-hoc* networks are not applicable for the use with modern WSNs. In [GMG07], the similarities and differences between WSNs and *ad-hoc* networks are discussed, and it commonly boils down to the different restrictions of the available resources, the transmission properties and use cases of these two domains.

Another difference between a WSN and a MANET is also the network size. Today, WSNs, are commonly used in Home and Building Automation (HBA) *Environmental Monitoring* and *Smart Agriculture* systems [SCK+14] [Laz13] [Kas20]. Their main purpose is to monitor the environmental conditions and transmit the measured samples to a central controlling system in these domains. Due to the wireless transmission, these measured data values are broadcast to all the nodes within a node transmission range. Therefore, as described in [RK18], efficient multicast protocols are significant for WSNs, e.g. to deliver reconfiguration messages to certain nodes without the need to address each node individually. Due to the high number of nodes used in modern WSNs, wireless *network simulators* like the ones discussed in [KHN14] are essential to develop and test such a network before deploying it with real hardware nodes. The usage of state-of-the-art *network simulators* allows the developers to skip the costly and time-consuming step of physically implementing the network and instead simulate specific use cases to analyse the energy consumption, performance, network behaviour even functional correctness of a completely virtual network.

Different open-source network simulators allow us to virtually deploy a WSN and simulate its behaviour. Minakov et al. and Bakni et al. provide a detailed comparison between the features and performance of the most popular simulators for WSNs [MPRS16] [BMC+19], like the famous NS-2, OMNeT++ and the *Contiki/Cooja* simulator [IH12] [VH08] [ODE+06]. The only simulator missing within these comparisons is the NS-3 simulator [RH10], which is developed as a successor for the discontinued NS-2 simulator which is no longer developed nor maintained. This NS-3 simulator is an improved and completely from scratch re-developed version, which is not compatible with the NS-2 simulator. NS-3 is enhanced with various additional simulation models compared to NS-2, is actively maintained and receives continuous updates as well as new features and simulation models.

Further, the work of Bakni et al. also aims to give a detailed view of how the different simulators implement their used energy models and what features and functional limitations these energy simulation models have [BMC+19]. As shown, all of the analysed simulators use the *state-based* approach to model the energy consumption of a node. Therefore, they directly monitor how much time a device spends in certain Physical Layer (PHY) states. This period is then multiplied with the supply voltage and consumed current to calculate the assumed energy consumption of a certain state. Each time the PHY changes its state, these calculated energy values are summed up and represent the total energy consumption of a node during the simulation. The newly developed NS-3 simulator and its energy model implementation use this *state-based* approach to calculate the overall energy consumption of a node [WNP11]. Depending on the actual simulator, the used energy simulation models further provide additional features, like the simulation of a connected battery, that is drained by the simulated energy consumption of the node and shuts down the whole node if its available energy is depleted.

1.4 Methodology and Structure of the Thesis

The thesis is structured and divided into multiple chapters, which handle this work's different aspects and methodology.

This thesis aims to evaluate and analyse the possibilities of simulating and comparing the energy efficiency of multicast routing protocols. We first must research available network simulators, check which features and energy measurement functionalities they provide us, and select the best suiting one. In chapter 2, we describe the work conducted for this thorough comparison and elaborate on our final decision to stick with NS-3. This chapter introduces how to use the NS-3 simulator and implement new NS-3 compatible models. Further, this chapter also explains the most critical software components of NS-3, how we use it to simulate WSNs and how to extend and contribute new simulation models to its code base.

Next, we need to pick a set of different routing protocols we are using throughout the whole thesis and which we want to further evaluate with a simulator. These selected routing protocols are *Mesh-Under Flooding*, MPL with both its *proactive* and *reactive* operating modes and the standard AODV routing protocol. In chapter 3 we provide a detailed description of the properties and mechanisms these selected routing protocols use.

Chapter 4 describes the necessary adaptions to the NS-3 simulator, such that we can later use it to simulate the actual use cases and experiments, from which we then extract our results. All of these necessary software implementations are listed in this chapter. It contains the description of the work done to implement the new simulation models for the MPL routing protocol, the Low-Rate Wireless Personal Area Networks (LR-WPAN) energy model, the porting of the AODV model to AODV6 and the development of a whole automatic *Simulation/Evaluation Framework*.

In chapter 5, we introduce the different simulation topologies and models we use for our simulations, as well as the different routing protocol configurations we are deploying to these simulation models. The simulated topologies described in this chapter are the *line*, *circle* and *grid* topologies.

Chapter 6 contains an evaluation and discussion of the results acquired from the simulations we performed. This chapter discusses what these simulation results provide, how they can be interpreted in the particular context of energy efficiency, and how the simulated results compare to actual hardware measurements.

Finally, chapter 7 concludes the thesis and provides an outlook on possible adaptions and extensions to our performed simulations and lists the future work.

CHAPTER 2

Wireless Network Simulators

Network simulators play a vital role in the realization and development of WSNs, as they provide the possibility to check and trim the behaviour of the nodes' utilized protocols by trimming specific parameters to optimize the overall performance of the WSN. Modern simulators allow us to simulate many different aspects of a wireless network. We can use their generated data to analyse and improve energy efficiency, throughput, reliability and other network properties. That can be done either by using different routing protocols or by fine-trimming, such that the used ones suit the desired use cases best. Due to the high importance of these simulators, there are quite a lot of them available, so we want to compare them against each other to provide a quick overview of them. This chapter introduces and discusses the following four simulators, which are open-source software, else it is not possible to use and adapt them for free, and commonly used to perform state-of-the-art network simulations.

These simulators are *Cooja* [ODE+06], which is generally used to simulate WSNs utilizing the well known *Contiki OS* as their base operating system. The second simulator on our list is *OMNeT++* [VH08] together with its specialization *Castalia* [NHH18], that especially designed for the simulation of WSNs. The last two discussed simulators are the *NS-2* simulator [IH12], and its modernized state-of-the-art successor *NS-3* [RH10].

2.1 Simulator Evaluation

First, we discuss the four previously mentioned simulators and list their strengths and weaknesses. We compare them against each other and select the simulator that is the best suited for the work done in this thesis.

When comparing the simulators against each other, there are three crucial properties that a simulator has to fulfill, in order to be further considered for use within this thesis. These properties are:

- **Actively Maintained:**
  The simulator has to be actively maintained, such that possibly implemented simulation models and protocols can be streamed back into the original source-code and possibly included into a future release. For a discontinued and outdated simulator, this further usage of our implementation work is simply not possible anymore.

- **Discrete Event Simulator:**
  As we want to use a state-of-the-art simulator, that achieves a high and scalable performance and calculates our simulation results in a minimum amount of time, it is essential that it uses a discrete event simulation engine.

- **Support for LR-WPAN (IEEE 802.15.4), 6LoWPAN and IPv6:**
  Our main goal is to simulate modern WSNs that could possibly be connected and routed into the IoT. The availability of these three simulation models is a key property, to keep the amount of additional implementation work low, because if one of these basic protocols is already missing, it is unlikely that there is support for any other IPv6-based routing protocol usable with WSNs.

2.1.1 Cooja/Contiki-NG

The first simulator is *Cooja*, which simulates the network behaviour of nodes that run *Contiki OS*. *Cooja* is programmed in *Java* and offers an intuitive and easy to use GUI. In Figure 2.1, an example of *Cooja's* GUI is shown. As seen in Figure 2.1, it provides the user with an overview of the network topology, the nodes' sent messages, and the bottom part shows the utilization and duty cycle of the nodes' radio interface. The GUI can be adapted easily, and extra functions and windows are available to depict additional properties of the simulation.

Due to its heavy dependency on *Contiki OS*, *Cooja* is tightly coupled with *Contiki* and commonly used to simulate nodes and applications that use it as their Operating System (OS). Apart from this tight coupling with *Contiki*, it is still possible to use *Cooja* to simulate nodes that utilize other OSs, like, for example, *RIOT OS*.

Due to this possibility to simulate different OSs, *Cooja* enables us to use a big variety of important WSN protocols, like the ones from *Contiki*, which features a 6LoWPAN, MPL and even a fully functional RPL [WTB+12] implementation. Additionally, *Contiki* also supports high-level application protocols and allows the user to customize the network stack. Therefore, *Contiki* and its simulator *Cooja* are a solid choice for developing an application for wireless networks for real-world applications and physical WSNs.

The only major problem with *Cooja* is its simulation engine, which is partly implemented as a Discrete Event Simulator (DEVS), while on the other hand, it is also vastly based on an explicit hardware emulator called MSP-Simulator (MSPSim) [EOF+09]. This emulator allows to exactly virtualize the behaviour of hardware nodes based on the *TI MSP430* microcontroller. One such example is the *Zolertia Z1* mote, which is most

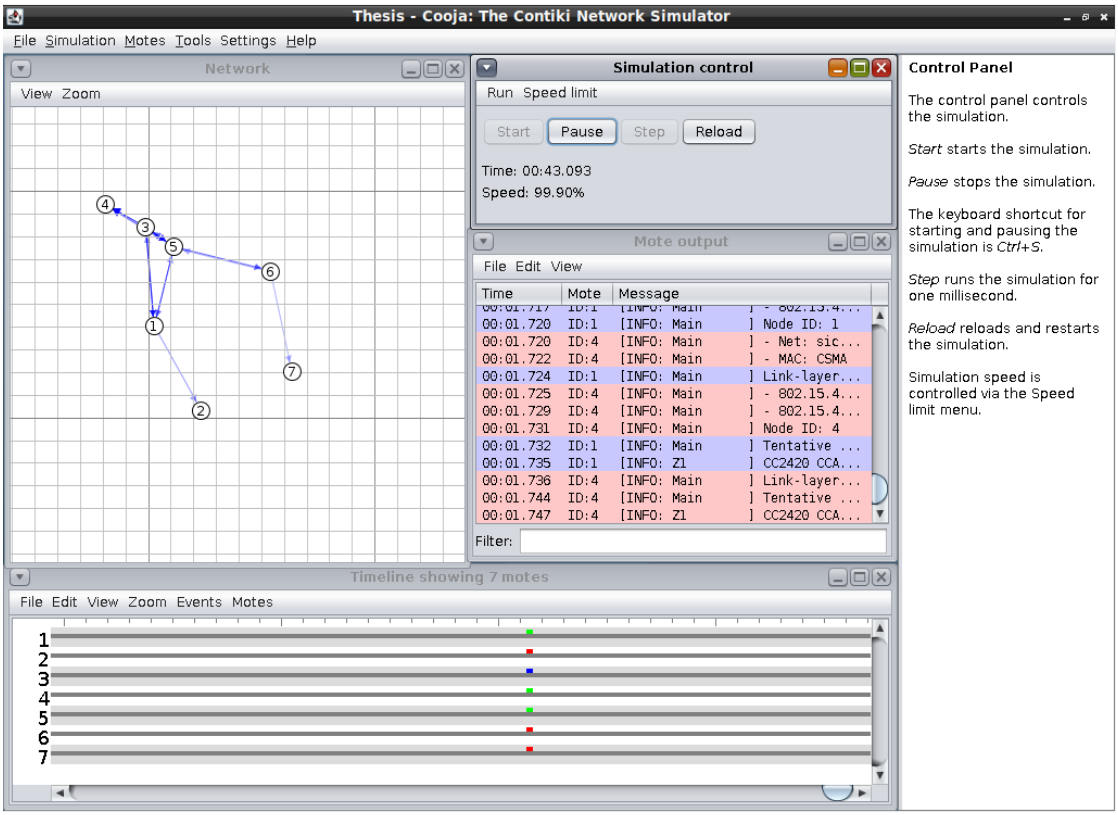


Figure 2.1: GUI of the *Cooja* simulator during an active simulation

commonly used to implement simulations in *Cooja*. Due to this hybrid combination of a DEVS together with an explicit hardware emulator for each node inside a simulation, its generability lacks and it is not suitable to simulate large heterogeneous networks. Instead, *Cooja* is often used to test specific application implementations and its main intention is not to analyse protocols for their exact behaviour in broad networks which typically needs a simulator whose performance scales better. A noteworthy feature of *Cooja* is its support of custom plugins that can be installed and used to simulate various network aspects. An example of such a plugin is the support for a mobility model, which is not part of the basic *Cooja* installation. Despite that, a mobility model is available online from the broad *Contiki* community, which has to be manually added to the *Cooja* installation to simulate movable nodes. However, another problem of *Cooja* also originates from its hardware emulation approach. This problem is its lack of advanced debugging mechanisms to step through a program and analyse it in detail. Instead, it only features a so-called *Event-Listener*, that pauses the simulation in certain situations. Debugging a program or protocol with such limited analysis tools can be very tedious, and discovering bugs and errors in the motes may be tricky.

The last problem that often occurs when using *Cooja* is that the only devices supported do

not have enough memory to run a specific application. In such a situation, *Cooja* cannot build and simulate the code. This behaviour may be good to know when developing a real-world application. Nevertheless, when using *Cooja* for theoretical simulations and protocol analysis, this type of behaviour can be bothersome because a user always has to keep track of the memory usage and switch the mote types if a particular node runs out of memory. In the worst case, creating a new hardware definition may be necessary if all supported and existing nodes do not have enough resources or peripheral connections.

Writing such new hardware definitions is quite a tedious process. We have to extend the suitable compiler toolchain to support the newly defined hardware device so that it can create functional binaries for the new virtual microcontroller. An additional drawback that originates from the utilization of the hardware-emulator is *Cooja's* lack of support for generic simulations. Due to the need to build a separate executable for each specific type of node, the available nodes are not capable of storing generically programmed binaries that support multiple routing protocols like those implemented in subsection 4.5.3. Such an implementation would most likely consume all the memory of a single node and render it useless. Therefore, *Cooja* is not suitable to perform parameter variation simulations of a large WSN. Each change of its configuration requires additional re-compilations and overhead to create the new binaries for the nodes for each configuration change. These time consuming intermediate steps reduce the scalability, simulation performance and usability of the simulator. This lack of scalability is a crucial disadvantage when performing a high number of simulations for academic and research purposes rather than designing and testing a specific application's use case.

2.1.2 OMNeT++/Castalia

The second network WSN simulator is the *OMNeT++* simulation framework [VH08], and its additionally required software components, which form the *OMNeT++/Castalia* network simulator. These additional software parts are the WSN simulator *Castalia* [NHH18], and the *INET* network simulation framework [MVK19]. *OMNeT++* is the generic, discrete event framework that provides the discrete simulation engine. This framework provides the required features, that are necessary to implement a functional network simulator on top of it. In this case, the simulator using the *OMNeT++* framework is the so called *Castalia* wireless network simulator. This simulator implements the GUI to design and configure the simulations and it triggers and controls the simulations that are actually performed by the underlying *OMNeT++* framework. The third component necessary to perform a network simulation with the *Castalia* simulator is the *INET* framework. This part contains the actual implementations of the different simulation models, like the LR-WPAN model, which *OMNeT++* and *Castalia* need to simulate a WSN. The *INET* framework provides a broad variety of different simulation models, for example there are mobility, energy consumption, transmission and routing protocol models that can be used to simulate any form of communication network, when using *Castalia* it would be a wireless network.

Like in the previous section with the *Cooja* simulator and *Contiki*, *Castalia* is also

absolutely dependent on the *OMNeT++* simulation framework and extensions containing the simulation models like *INET* or other custom software part that implement missing simulation models. The software architecture of *OMNeT++* requires this strict separation into these three different software parts, that are needed to perform a simulation. These parts are the general *OMNeT++* framework implementing the discrete simulation engine, a library with the network simulation models like *INET* and a simulator like *Castalia* that implements the GUI and controls the whole simulation process. Due to this strict separation, each part is a specialization with a specific purpose and provides the user with an excellent tool-set to perform analytic and realistic simulations without a specific hardware platform or processor. *Castalia* and *OMNeT++* are both programmed in C++ and are commonly used for academic simulations and protocol analysis. Due to their high scalability, the number of supported simulation models and the high amount of core functionalities to analyse and visualize the simulation results.

The only problem with this simulation framework is the currently lacking support for WSN protocols as there is no working version of the 6LoWPAN adaption-layer protocol and other IPv6 based routing protocols that can be used with the existing *IEEE 802.15.4* model. Kirsche and Schnurbusch present a very impractical and inefficient workaround to use the precompiled 6LoWPAN implementation from *Contiki* [KS14], with the *OMNeT++* simulation engine. Using this workaround has several drawbacks that originate from using a model that is precompiled for the use with *MSPSim*. In order to use this precompiled binary, the proposed 6LoWPAN model implements a wrapper that emulates the behaviour of *MSPSim* to use the binary with *OMNeT++*. Further, this approach of reusing the implementation from *Contiki* is bad for the overall performance, scalability and especially its usability, as all the protocol configurations have to be done before compiling the model in *Contiki* and cannot be changed within *OMNeT++*. Due to these shortcomings, this workaround is not contained in the official version of *INET* and can only be used as an unofficial extension-plugin together with the very old *OMNeT++* framework version 3.4. As the currently released version of *INET* framework is 4.3.7 and this 6LoWPAN plugin is not maintained anymore, it is incompatible and dysfunctional with the latest versions of *OMNeT++* and *Contiki OS*. So there is only a single possibility to use this model. One has to use the old *OMNeT++* version where the plugin is functional and misses out on all the new features *OMNeT++* offers. Therefore, this prevents us from using models and functionalities added in subsequent versions, only to have a model for 6LoWPAN. This fact renders it not reasonable to use the stated 6LoWPAN model in a state-of-the-art simulation environment. Other than the generally lacking support for WSNs, the whole *OMNeT++/INET/Castalia* software packet features a broad range of available protocol implementations to simulate wireless networks apart of WSNs, like *WiFi* networks, and achieves realistic first order validation simulations of protocols without actual hardware and any further loss of generality.

2.1.3 NS-2

Another widely, if not the most used network simulator is *NS-2*. Like *OMNeT++*, *NS-2* is also a discrete-event simulator, which allows fast and deterministic simulations of both wireless and wired networks. *NS-2* is still commonly used for scientific purposes and academic research, which is quite impressive, as the initial release of *NS-2* was in 1996, and active development officially ended in 2011 to focus the work on its successor *NS-3*. Like the previous simulators, *NS-2* features various models that are necessary to implement a realistic and accurate network simulations. The usage of *NS-2* is quite different than with the other previously introduced network simulators. Instead of a comfortable GUI, this simulator can only be used by programming the models and simulations in C++ or the specialized scripting language OTcl, which is an adaption of the basic Tcl script language. That means the user has to write a C++ or OTcl program by using the available simulation modules as a library to implement the behaviour of the nodes. This program is then compiled and started to execute the simulation.

Due to the complete lack of a GUI, NS-2 provides neither a graphical designer to create the simulation network nor a visualization of steps happening during the running simulation, like it is available with *Cooja* or *Castalia*. The *NS-2* simulator only has an optional software part that provides a visualization of the network behaviour after the simulation has finished. Therefore, the simulation process writes an optional log file that contains all the necessary information to visualize every single simulation step in a separate visualization software afterwards. The additional network animation software then reads these written events from the log file and visualizes them in a minimalistic GUI.

The advantage of *NS-2* in comparison to other simulators is that there are no additional frameworks and additional software parts needed to perform a simulation. An *NS-2* installation contains all of the official *NS-2* simulation models right from the start, without any additional software frameworks or components, and the user can choose and use the models needed for simulation. Another strength of *NS-2* is that you can use all the features of C++ to implement your simulation and use other additional C++ libraries to visualize the results of a simulation.

Of course, there are also drawbacks when using the *NS-2* simulator, is the limited availability of IPv6 based protocol, and the lack of an *IEEE 802.15.4* and 6LoWPAN model, which are absolutely crucial for the WSN simulations we want to perform. Another major problem and showstopper for *NS-2* is, that its development was already stopped in 2011 is not continued anymore, which renders NS-2 completely outdated.

2.1.4 NS-3

As mentioned in the previous section, *NS-3* is the improved and adapted version of the successful *NS-2* simulator, so it inherited most of its properties from *NS-2*. In comparison to its predecessor, *NS-3* can be programmed with the standard languages C++ and Python without any further need of OTcl. This switch of the supported programming languages was possible due to a complete redesign and reimplementation of

the *NS-3* core, which was necessary to acquire a modern and fast simulation environment that offers the same features as *NS-2*. Although *NS-3* was completely re-implemented and renewed, its usage and essential features are quite the same as with *NS-2*. One big difference between the two simulator generations is the high number of additional simulation models that *NS-3* offers. These newly implemented models feature broad support of IPv6 based protocols like 6LoWPAN and additional transmission layers like LR-WPAN, which enables our simulations to use the *IEEE 802.15.4* protocol. These new models allow a user to perform state-of-the-art simulations of IoT networks. Another advanced feature of *NS-3* is the possibility to use it for *hardware-in-the-loop* simulations, which allows the simulator to interfere with real hardware and act as a testbench that tests the network communication of the hardware.

2.2 Simulator Comparison

Now that we introduced some commonly used network simulators, Table 2.1 visualizes their most important properties in detail. It also lists the available simulation models of each simulator that are most important for this work. The key models are *IEEE 802.15.4* and *6LoWPAN* as these two models are mandatory to simulate modern WSNs.

	NS-2	**NS-3**	**Castalia/OMNeT++**	**Cooja/Contiki**
Simulator Type	Discrete-Event	Discrete-Event	Discrete-Event	Hardware Emulator
Language	C++, TCL	C++, Python	C++	C
GUI-Support	Poor	Poor	Very Good	Very Good
Scalability	Very Good	Very Good	Very Good	Poor
Documentation	Very Good	Very Good	Good	Poor
Simulation Models	☑IEEE 802.11, ☐IEEE 802.15.4 ☐6LoWPAN ☑AODV, ☐MPL ☐RPL ☐(E)-SMURF ☑Mobility, ☐Energy model, ...	☑IEEE 802.11, ☑IEEE 802.15.4 ☑6LoWPAN ☑AODV, ☐MPL ☐RPL ☐(E)-SMURF ☑Mobility, ☑Energy model, ...	☑IEEE 802.11, ☑IEEE 802.15.4 ☐6LoWPAN ☑AODV, ☐MPL ☐RPL ☐(E)-SMURF ☑Mobility, ☑Energy model, ...	☑IEEE 802.11, ☑IEEE 802.15.4 ☑6LoWPAN ☐AODV, ☑MPL ☑RPL ☑(E)-SMURF ☑Mobility, ☑Energy model, ...
Limitiations	No new simulation models No IEEE 802.15.4	No RPL, IPv6 multicast support, No energy model for IEEE 802.15.4	No mobility model No 6LoWPAN, No RPL, Multiple component libraries	Lack of Debug possiblities, Memory limitations of nodes Relatively slow simulations Bad scalability, Hardware compilers necessary No generic simulations ...
Special Features		Hardware-in-the-Loop	Excellent simulation framework	Contiki-OS simulations
Actively Maintained	No, Last Release November 2011	Yes, Last Release October 2021	Castalia: No, Last Release October 2013 OMNet++: Yes, Last Release November 2021	Yes, Last Release July 2021

Table 2.1: Comparison of the introduced simulators

As this table shows, *NS-3* and *OMNeT++* are the most versatile simulators as they use a state-of-the-art simulation framework and offer a broad range of available simulation models. Though, they both have a common limitation, by not supporting a simulation model for the RPL protocol. The only drawback that *NS-3* has in comparison to

OMNeT++ is the very minimalistic GUI support and visualization of the simulation. There is only a rather trivial network animator in *NS-3* that provides a rough overview and analysis of the wireless network. So, if there is any specific visualization or diagram output needed, a user has to implement such applications himself with all the available C++ and Python libraries. Although, *NS-3* still supports some additional models, and its overall usage and documentation are very good in comparison to *OMNeT++/Castalia*. Another important detail of *NS-3* is that all of its components are actively maintained, and there are periodic improvements and updates of the whole simulator available. The other simulators like *NS-2* and *Castalia* are not maintained anymore. For the *Castalia* simulator, only the *OMNeT++* base framework features active development and maintenance. While the implementation and code of the *Castalia* simulator that is used on top of *OMNeT++* and *INET* was last updated in 2018.

Cooja on the other hand, uses a different simulation core than the other three introduced simulators. These use a fast and efficient discrete event simulation engine that allows skipping idle periods during a simulation, which speeds up the whole simulation process and generates excellent scalability to use the simulators in simulations with up to hundreds of different nodes, as *Cooja* uses a hardware emulation engine to simulate the processor and network of the nodes. So the simulator has to emulate each node for itself, which is very bad for the performance. Instead, it even has to emulate the waiting and idling of the processors if they have to wait until a specific event occurs. Therefore, the simulation engine of *Cooja* is not scalable to simulate bigger networks in a reasonable amount of time.

We can conclude from the simulator comparison that *NS-3* and *OMNeT++/Castalia* are the most advanced ones that offer a broad range of features. However, when it comes to the important details, the latter simulator does not feature a currently working 6LoWPAN model, which eliminates it from our available choices. For the same reason *NS-2* also cannot be used. Therefore, we can only select between *NS-3* and *Cooja*, which both feature all the mandatory models. *Cooja* has the advantage of various additional routing protocols in comparison to *NS-3*. Nevertheless, this single advantage cannot outweigh the various drawbacks due to the limited scalability, debug-ability and other limitations listed in Table 2.1.

2.3 The NS-3 Simulator

Due to the comparison in Table 2.1, the available simulations models, general usability and extend-ability of the simulators, NS-3 is chosen for further work in this thesis. NS-3 is generally very well documented, the documentation and simulator can be obtained from its official website[1].

Additionally, NS-3 also provides a centralized documentation of all the officially supported simulation modules and the different functionalities that each of the models provides.[2].

[1] Official NS-3 website: `https://www.nsnam.org/releases/ns-3-34/`
[2] NS-3 simulation models documentation: `https://www.nsnam.org/doxygen/modules.html`

2.3.1 NS-3 Core Features and Functionalities

Since NS-3 is the successor of the famous and commonly used NS-2, it aims to add further improvements and additional features to allow simple and easy-to-use simulations. Therefore, it has various specialized features and functions to achieve these goals.

For an overall understanding, we first have to introduce the mandatory *Core* module and why we need it. This essential module is the heart of NS-3 and implements the whole simulation framework, which is necessary to perform a simulation run. The functionality of the *Core* module can be compared with the one of *OMNeT++*, with the difference that it is completely integrated into the source-code of the simulator and not a separate and stand alone software component.

The following main functionalities of the *Core* module were used to implement the different simulation settings and new simulation models described in chapter 4.

- **Simulator**:
  This functionality is the heart of NS-3, and all the other simulation modules depend on this mandatory module. This *static* class creates, starts and destroys the actual simulation process, keeps track of the internal simulation time and stops the simulation after it reaches its end-time. Each simulation has to use the different functions this `Simulator` class provides to control the overall simulation process and schedule additional events during an active simulation.

- **Smart Pointer**:
  The NS-3 *Core* module features a specific and non-standard dynamic memory management system that takes care of the object creation and destruction. This custom *Reference Counting* system handles the automatic deletion of unused memory objects. This system keeps track of the dynamically created class objects, detects if there are no more references to an object, and automatically deletes it for the user. This functionality simplifies the implementation as the programmer does not have to care for the correct deletion of all the created objects to prohibit memory leaks and free all the used memory. Therefore, the *Core* module implements special template classes and specific usage of how to work with this self-created *Reference Counting* mechanism.

- **Attributes**:
  The NS-3 core does not only feature its own and special `Smart Pointer` implementation. Further, the `Core` module has a specific Application Interface (API) to define an attribute handler to access the *private* properties of the classes. These handlers are custom *Setter*-methods used to assign a value to a certain class attribute. For example, this allows implementing a *Setter* method that checks if a specific value is correct and is assignable to a class attribute. This behaviour allows the implementation of value checkers that catch out-of-bound values before assigning them to the internal class attribute. The only requirement for this mechanism

to work is an inheritance from the `Object` class for the new class. Due to this inheritance, the new class inherits the implementation of method `GetTypeId (void)`. This function creates a new static object of the `TypeId` class, where we have to define the classes properties, for the usage with the *Smart Pointer* system of NS-3 and where we can add our custom attributes that shall be accessible via this mechanism. Listing 2.1 shows an example definition of a class using the *Attributes* functionality. The shown code defines the class *ClassName* and calls the member functions to perform some mandatory settings by specifying the classes inheritance parent, its member group name and the classes constructor. Without this information, the *Smart Pointer* functionality does not work as intended. Afterwards a single unsigned integer attribute that is initialized with 1 and the property `m_InternalClassProperty` is checked and filtered for a applicable value by the generic `UintegerChecker` class.

```
1        TypeId ClassName::GetTypeId (void)
2        {
3            static TypeId tid = TypeId ("ns3::customNamespace::ClassName
    ")
4            .SetParent<InheritanceParent> ()
5            .SetGroupName ("GroupName")
6            .AddConstructor<ClassName> ()
7            .AddAttribute ("AttributeName", "String describing the
    Atrribute.",
8            UintegerValue (1), // The init-value
9            MakeUintegerAccessor (&ClassName::m_InternalClassProperty),
10           MakeUintegerChecker ()); // The sanity checker-method
11           return tid;
12       }
13
```

Listing 2.1: Example of an Class and Attribute Definition

Now that we defined a custom class and its attributes, we can access and manipulate the attribute's value with the listed functions.

```
GetAttribute ('AttributeName', v_Variable)
```

```
Set ('AttributeName', UintegerValue (100))
```

These functions allow us to access and change the hidden private properties of a class everywhere we need them without giving away complete control of the property itself via the indirect *Setter* functions.

A detailed description of the attribute frameworks functionality is given in the official documentation of NS-3[3].

- **Command-Line**:
  Another commonly used functionality of NS-3 is its integrated parsing of command-line parameters issued to the simulator throughout the *waf* build system. This

[3]Documentation of the Attribute-Framework: `https://www.nsnam.org/docs/manual/html/attributes.html`

mechanism allows users to configure the simulation at startup without recompiling the code, offering the possibility to implement reconfigurable and generic models. Users first define the possible arguments and their respective variables to use this functionality to store the values. Next, the `Command-Line` automatically searches the passed caller arguments for the defined keywords and parses their values into the specified variable. Listing 2.2 shows the using of the argument parsing. This code initializes the command-line parsing and adds a possible argument that shall be parsed from the command-line by the following `Parse (argc, argv)` function. This method checks all the passed arguments for occurrences of the declared command line arguments, and if this is the case, it stores its value into the intended variable.

```
1        CommandLine cmd(__FILE__);
2        cmd.AddValue("argumentName", "Description of the argument",
    argumentVariable);
3        cmd.Parse(argc, argv);
4
```

Listing 2.2: NS-3 Argument Parsing Example

- **Pseudo-Random Simulations**:
  Another mandatory feature of a state-of-the-art network simulator is to guarantee pseudo-random simulations. These simulations allow us to calculate deterministic results if a simulation is re-executed with an identical random-seed configuration as the previous one. Therefore, the simulation framework has to support and implement a seeding process for its internal random generators, which generate the internal random variables, enabling two simulations with identical random seeds to achieve the same results. In NS-3, this pseudo-randomness can be handled with the `RngSeedManager` class, which allows the user to initialize the random-number generation to ensure deterministic simulation results.

  The `RngSeedManager` differentiates between two different types of seeding values, the actual *random-seed* and the *run-number*, which is used for multi-run simulations to introduce some randomness without to change the actual *random-seed*. When recalculating the same result of a certain simulation, both of these values have to be configured correctly with the static functions `SetSeed(seed)` and `SetRun(run)` when starting the simulation before it uses all its internal random variables for the first time. After these steps and a correct initialization of the Random Number Generator (RNG) and its `RngSeedManager`, we can re-create previously calculated simulation results.

- **Variable Tracing**:
  With the `Variable Tracing` functionality, NS-3 allows us to track every single change of a special variable. Every time the value of the tracing variable is modified, this triggers the execution of a defined callback function, enabling this feature to implement various statistics and evaluations, like counting the number of sent packets by a certain node. Further, `Variable Tracing` is useful to debug

applications by tracing and printing every assigned value. So it provides a good overview of all the different values a property takes at runtime without using a debugger and iterating through the application.

- **Trickle Timer**:
  At last, we discuss one of the newest features of the `Core` module, which is the implementation of the `Trickle-Timer` functionality. As the name suggests, this is a fully functional implementation of the *Trickle* algorithm [LCH+11]. This feature is mandatory for this work as the whole MPL protocol roots on the `Trickle-Timer` and its behaviour. The NS-3 framework provides us with this implementation that allows us to define a callback function, which it executes after the integrated timer has fired, to trigger the further handling of this occurred event. Additionally, the implementation provides all the necessary functions to start, stop, reset and issue consistent and inconsistent events to the timer. This implementation handles all of the various edge cases of a `Trickle Timer` and the correct firing of the callback function.

2.3.2 Building, Executing and Debugging a Simulation

This section will look at the general usage of NS-3 and which working steps besides the installation process are necessary to implement, execute and debug a simulation such that it behaves as intended. This section introduces these essential tasks when working with NS-3.

Implementing an NS-3 Simulation

After successfully installing NS-3, the first step is implementing a custom simulation. The first and obvious step is to define the simulation scenario and implement it with the available models provided by the simulation framework. For help with the necessary programming work, there is documentation and example code in the *Examples* folder of NS-3 itself or inside the model's source code directory. This code can be used as a template and combined with other available simulation models. By combining various models it is possible to implement an scenario -- in our case using the LR-WPAN-, 6LoWPAN-, *Internet*- and *Application*-modules -- it is possible to implement a simple WSN, which sends ping messages from a source to a sink node.

After the implementation of our custom simulation model is finished, we have to configure the *WAF* build system to include our new code into the build process. Therefore, it is necessary to adapt the *wscript* file within the same folder as our newly created simulation. Listing 2.3 shows the *wscript* file of the two simulations contained into the *Energy-Examples* of NS-3. As this example shows, two lines for each simulation define the simulation name and its necessary dependency models. The second line lists the files containing the source code of the simulation we want to build.

```
1  ## -*- Mode: python; py-indent-offset: 4; indent-tabs-mode: nil; coding:
   utf-8; -*-
2
3  def build(bld):
4  obj = bld.create_ns3_program('energy-model-example', ['core', 'mobility',
    'wifi', 'energy', 'internet', 'config-store'])
5  obj.source = 'energy-model-example.cc'
6
7  obj = bld.create_ns3_program('energy-model-with-harvesting-example', ['
   core', 'mobility', 'wifi', 'energy', 'internet', 'config-store'])
8  obj.source = 'energy-model-with-harvesting-example.cc'
```

Listing 2.3: Example of a *wscript* file

Executing an NS-3 Simulation

Now that we have successfully implemented our custom simulation and configured the build system to use it, we can assemble the new simulation. All that is necessary to achieve this is to issue `./waf --run=energy-model-example` at the root directory of NS-3, this will start the build process of the selected simulation by recursively resolving and building all the defined dependencies, then it generates an executable binary file that contains the execution, which is then automatically executed to perform the simulation.

This example shows the modularity and simplicity of building and executing a simulation in NS-3. Another noteworthy feature of this build process is the advanced dependency resolution. This functionality allows simplifying the definition of necessary code dependencies.

Another feature of NS-3 and its *WAF* build-system is its support for automatically executing the simulation's binary inside a debugger. Hence, the developer can use this extended scripting mode to debug the program and detect possible problems during runtime.

2.3.3 Developing Custom NS-3 Models

Now that we have a general understanding of the usage of NS-3, we will discuss the components necessary to implement a new simulation model and integrate it into the *WAF* build-system. In Figure 2.2, the structure and different files of the NS-3 6LoWPAN simulation module are shown. This directory scheme is fixed for all the models and ensures that the build system can detect all the models and integrate them into the finally built executable.

So the first step when trying to develop a custom simulation model is to create the basic model structure, which is needed to build and run a simulation that utilizes this new module later. To simplify this initial creation of the correct folder and file structure for a new model, NS-3 contains the `create-module.py` script in the `ns3-dev/utils` folder performs this initial work and creates a basic skeleton of our new module.

```
sixlowpan-module
├── wscript
├── test
│   ├── sixlowpan-iphc-test.cc
│   ├── sixlowpan-iphc-stateful-test.cc
│   ├── sixlowpan-hc1-test.cc
│   ├── sixlowpan-fragmentation-test.cc
│   ├── sixlowpan-examples-test-suite.cc
│   ├── mock-net-device.h
│   ├── mock-net-device.cc
│   └── example-ping-lr-wpan.reflog
├── model
│   ├── sixlowpan-net-device.h
│   ├── sixlowpan-net-device.cc
│   ├── sixlowpan-header.h
│   └── sixlowpan-header.cc
├── helper
│   ├── sixlowpan-helper.h
│   └── sixlowpan-helper.cc
├── examples
│   ├── wscript
│   ├── example-sixlowpan.cc
│   ├── example-ping-lr-wpan-mesh-under.cc
│   ├── example-ping-lr-wpan.cc
│   ├── example-ping-lr-wpan-beacon.cc
│   └── CMakeLists.txt
├── doc
│   └── sixlowpan.rst
├── CMakeLists.txt
└── bindings
    ├── modulegen__gcc_LP64.py
    ├── modulegen__gcc_ILP32.py
    └── callbacks_list.py
```

Figure 2.2: The code structure of the 6LoWPAN simulation model

Then, we have to declare and configure the *WAF* build system to include these new source files in the build process. To do so, we again have to edit the `wscript` inside the directory of the new module and declare the modules *module-dependencies*, *soure files*, *header-files*, *unit-test files* and additional *example simulations* inside of the new module folder. There are also further and optional declarations, like the *helper*-classes that can be implemented to simplify the assigning of such a module-object to a specific node within the simulation. As shown in Figure 2.2, optional documentation files and python-bindings allow us to use the code written in C++ in combination with a Python script.

These declarations are similar to the ones shown in Listing 2.3. The only difference is that this `wscript` file contains the code to build the source code and examples inside of a simulation module and not to execute a certain simulation, as it is for the given `wscript` file in Listing 2.3. The official NS-3 documentation[4] provides a detailed description on how to correctly write and define all mandatory and optional code parts in a `wscript` correctly.

After this work of configuring *WAF*, we can now finally build and execute an example simulation that uses our newly defined and implemented module by issuing the commands previously described in section 2.3.2.

[4]Creating and Declaring a Custom NS-3 Module:
`https://www.nsnam.org/docs/manual/html/new-modules.html`

CHAPTER 3

WSN Routing Protocols

Due to the limited resources and low-power wireless transmissions of nodes in a WSN, they need specific routing protocols that are very energy efficient and do not exceed an inevitable transmission delay from the sender to all the designated receivers. Since all the transmissions in a WSN are wireless and therefore intrinsically broadcast to every receiver within its range, these routing protocols try to take advantage of this intrinsic broadcasting to achieve a reliable transmission over multiple hops while only using a minimal amount of energy.

There are various different routing protocols for WSNs we will look at and analyse them for their capabilities, like the support for multicast, their relevance in the domain of WSN, their implementation complexity and their availability within the NS-3 simulator.

The protocols considered and analysed are the commonly used *Flooding* protocol, multicast capable variants of the LEACH routing protocol, as well as RPL based multicast variations like Stateless Multicast RPL Forwarding (SMRF) and its numerous enhancements, the MPL routing mechanism as well as the original AODV protocol and its simplified variant called Lightweight On-demand Ad hoc Distance-vector Routing Protocol—Next Generation (LOADng) and numerous more. From this list we picked of three protocols, consisting of *Flooding*, *MPL* and the *AODV6*.

These three protocols are used in the subsequent chapters to implement, simulate and analyse for their energy efficiency and reliability in the domain of WSNs. As this thesis has a special focus on multicast routing protocols, we selected *MPL* and *Flooding* as both of them are capable to support multicast transmissions and MPL is even especially developed to achieve energy efficient and reliable multicast transmissions. In comparison we also choose the solely unicast capable *AODV6* protocol to compare its performance against the two multicast protocols, such that we can compare the energy effectiveness in similar use cases and determine if it can be reasonable to use a unicast protocol to

transmit multicast messages, by sending unicast messages to all the nodes within a multicast domain.

3.1 Mesh-Under Flooding

At first, we will look at the *Flooding* protocol to introduce it and evaluate its usage and some properties it features. The base version of the *Flooding* protocol is one of the simple and basic routing protocols available for WSNs. As stated in [BMLG17], the protocol is often used together with the *IEEE 802.15.4* transmission standard to implement a simple and resource-efficient routing mechanism to transmit packets via multiple hops across a wireless network.

As the name suggests, the *Flooding* protocol uses controlled broadcasting of packets across the whole network to transmit messages to their designated destinations. So each node repeats a received packet in the best case exactly once and sends it to all its neighbour nodes. Although, this best case cannot be guaranteed all the time, which is one of the main problems of protocols that utilize this approach. The neighbours receiving the message will also repeat it to reroute it to all of their neighbours. This procedure continues throughout the whole network. With this mechanism, the protocol aims to transmit the message to every reachable node inside the network. Due to the automatic broadcasting of wireless transmission, *Flooding* can route unicast and multicast messages without the need for another routing protocol, as every sending of a packet is automatically a broadcast transmission that floods across the whole network. So every node can decide if it has to drop or pass the message up to the next layer of its network stack.

There are some additional mechanisms in the *Flooding* protocol to prevent broadcast storms, which originate from uncontrolled forwarding of every received packet without checking if the node already routed the message before. So every node has to maintain a list of its currently handled messages to prevent further retransmissions of the same messages over and over, which would cause uncontrolled message floods, called the *Message-Cache*. If a node receives a new message, it stores it into its *Message-Cache*, such that in case it receives the same message again, the node detects that it already knows this message and does not forward it again. Additionally, as this *Message Cache* cannot store an endless amount of messages, there is also the TTL field in the packet header, which decrements at every hop. So this counter sets the maximum number of transmission hops a packet can make before it is not forwarded anymore by the nodes receiving it. This hop-counter is sometimes crucial to prevent a broadcast storm. Such a situation occurs if there is too much traffic inside the network and a node has already overwritten its complete message cache, which is limited due to the low resources of a node, with other messages. If the node then receives a duplicate of this message again, it forwards this again into the network, which then creates even more traffic and broadcast storm is developing and draining the nodes' batteries.

Flooding also has a third function to improve its energy efficiency and transmission

reliability, the so-called *Mesh-Jitter* timer of every node. This timer generates a random milliseconds delay that has to elapse before a received packet is forwarded again. This delay is necessary as the Medium Access Control (MAC) layer of *IEEE 802.15.4* uses CSMA/CA to access the shared transmission medium. CSMA/CA means that every node listens on the medium and checks if there is an ongoing transmission or if it is free and can be acquired. The *Mesh-Jitter* delay shall reduce the number of message collisions, as not all the nodes start to send at the same time as soon as the medium is accessible. This mechanism improves energy efficiency and reliability by reducing the number of collisions and the resulting retransmissions. The only drawback this *Mesh-Jitter* introduces is increased latency and delay for each transmission, which increases with every hop the packet makes, so the additional *Mesh-Jitter* delays are also summed up.

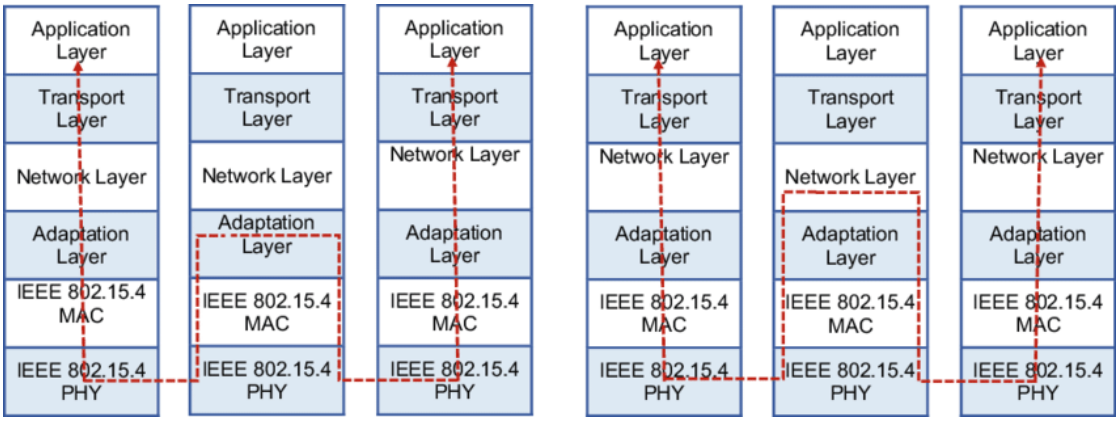


Figure 3.1: Mesh-under and Route-over network stack transitions [AK19]

Flooding can be used in two different ways inside a nodes network stack, either via the conventional way as a *route-over* protocol inside the *Network Layer*, on top of IPv6 and its specific adaption protocol 6LoWPAN. On the right part of Figure 3.1, the *route-over*, stack traversal is shown. For this type of routing, the *network layer* of the nodes network-stack implements the routing protocol. On the other hand, the left side of Figure 3.1 shows the *mesh-under* approach of routing protocols used in this thesis to simulate the *Flooding* protocol. With this approach, it is possible to route and forward the received packets on a lower layer of the network stack, which introduces some differences in the behaviour to the previously mentioned *route-over* routing. As [AK19] states the main difference between these two approaches is the handling of fragmented packets and the resulting energy efficiency and reliability. When using *Mesh-under* routing, each possible fragment of a whole message is handled and routed on its own. Therefore, if one such part is received improperly, only this single segment is dropped and needs to be retransmitted. In comparison, with a *route-over* routing approach, all the fragments have to be received correctly before the stack passes the message to the *network layer*, where the message is further processed. Therefore, if one such fragment is missing, all its correct received parts are also dropped and need to be retransmitted again. This behaviour is due to the routing in the *network-layer*, which processes only the complete messages as the

lower layers of the network stack already handle the fragmentation and defragmentation process. According to [AK19], the *route-over* mechanism has improved transmission reliability as there are more re-transmissions as with the *mesh-under* approach. However, due to the excessive number of retransmissions, this increased reliability also causes a much higher energy consumption, which again shortens the functional lifetime of the whole network when a high number of packet fragments are transmitted. Therefore, as the *mesh-under* routing is expected to achieve a better energy consumption than the classical *route-over* approach, we will use the *mesh-under* approach for the *Flooding* protocol in the further work of this thesis.

3.2 MPL

The second routing protocol analysed in this thesis is MPL, whose functional details are specified in *RFC 7731* [HK16]. Due to MPL being a multicast routing protocol, each node internally starts with subscribing itself to its desired IPv6 multicast domains and listens for messages destined to these destination addresses. After a node has successfully subscribed, it handles all the messages originating from these domains and tries to route them inside the wireless network.

The biggest difference between the MPL and *Flooding* protocols is that apart from forwarding recently received messages, it allows the nodes to retransmit packets multiple times automatically. Further, it also uses so-called *MPL-Control-Messages*, which list the data messages already known to a node. Therefore, MPL features multiple mechanisms to detect and counter the loss of a packet to achieve reliable transmissions. This increased reliability may again cause a higher energy consumption per node due to the additional retransmissions. Additionally, MPL is a typical *route-over* routing protocol, whose decisions are made in the *Network Layer* as it is depicted on the right side of Figure 3.1.

In the following subsections, we discuss the main parts of MPL, which are its two operation modes and the basic *Trickle* algorithm [LCH+11].

3.2.1 Trickle Timer

The whole MPL routing protocol originates from its usage of the *Trickle Timer*, which is an important algorithm used within MPL, that controls when the sending of a message shall be triggered. Every single message that has to be handled and transmitted by a node is associated with its message-specific *Trickle Timer*. This timer indicates and triggers the sending and retransmission of its associated message to the neighbouring nodes.

The algorithm itself is specified and described in [LCH+11]. Therefore, we will repeat the key aspects and set them into the specific context for the implementation of MPL according to [HK16].

A *Trickle Timer* is an event triggered algorithm, which defines two types of events. These two event types are the *consistent* and *inconsistent* events. These events are used

to interact with a running *Trickle Timer* and either signalize to it that everything is good and it does not have to fire soon, by sending a *consistent* event. Hence, sending an *inconsistent* signalizes the exact opposite to the *Trickle Timer*, as it then has to reconfigure itself and fire as soon as possible.

In general, a single *Trickle Timer* has three following parameters:

- $I_{\min}$ $[ms, s]$ is the minimum duration the timer has to be running. Its value range for usage with MPL is generally in milliseconds or seconds.
- $I_{\max}$ $[int]$ specifies the maximum duration the timer is allowed to run. This parameter is not a time value, instead it is an integer value which gives the number of doubling of $I_{\min}$. The following formula is used to calculate the maximum duration of a *Trickle Timer*:

$$t_{\max} = 2^{I_{\max}} * I_{\min} \tag{3.1}$$

- k $[int]$ is the redundancy constant of the timer and indicates the target value of the received *consistent* events during a single timer interval.

And the following three variables are used to implement the desired functionality:

- I $[time]$ is the current interval size and has to be within the inclusive borders defined by the two previously introduced parameters $I_{\min}$ and $I_{\max}$.
- t $[time]$ is a the *Trickle Timer's* current end time. Therefore, t is the time the timer is running. Its value has to be within $[\frac{I}{2}, I]$, which is calculated from the current interval time-span I.
- c $[int]$ is the *consistent* event counter. Each time such an event is received, this counter is increased.

Now that we have discussed all the parameters and variables used by a *Trickle Timer*, we will focus on the functionality and steps of the algorithm, as stated in [LCH+11]. As the first step when the algorithm is started, the *Trickle Timer* randomly selects an initial value for its interval-parameter I inside of $[I_{\min}, 2^{I_{\max}} * I_{\min}]$ and resets its internal counter c to 0. Additionally, it also initializes the variable t, a value from the interval $[\frac{I}{2}, I]$ and starts the timer with the value form the variable t. While the timer is running, we count the occurrences of *consistent* events during this interval. After the timer has reached its end value and we received less than k consistent events, the timer fires, else it continues without firing. Additionally, after the timer has finished, the current interval I is doubled with a maximum value of $I_{\max}$, and the timer can be restarted. A special case is the reception of an *inconsistent* event. This event resets the *Trickle Timer* instantly, I

is reset to $I_{\min}$, c becomes zero again, and t is also set to $I_{\min}$ and the timer is instantly started again.

Therefore, the algorithm does not instantly react to such an *inconsistent* event. Instead, the node sends its earliest response after $I_{\min}$ has passed. The node might not even answer to this *inconsistent* event at all if it receives enough *consistent* events during this short time. This delayed behaviour prevents instant firing from all the nodes that simultaneously received such an *inconsistent* event. Such a case would cause all the receiving *Trickle Timers* to fire and execute their respective expiration, which may generate a bottleneck situation.

The second property of a *Trickle Timer* is the counting of the *consistent* events. This behaviour introduces and implements a so-called *polite-gossiping* scheme. Therefore, a node is not a dumb repeater anymore that forwards everything it receives. Instead, it waits for a specific time and handles the occurring *consistent* and *inconsistent* events, to determine if a message has to sent or not.

3.2.2 MPL Message Types

In this section, we will now introduce and discuss the two types introduced by the MPL routing protocol. These are the *Data Message* with the special *MPL-Option* and the *Control Message*.

3.2.3 Data Messages

First, we introduce the *Data Messages* used to transmit application data. To generate such a *Data-Message*, we have to modify a basic IPv6 data packet, by adding an additional *MPL-Option* to the IPv6 header.

Therefore, we first have to recap how an IPv6 header is structured [DH98], how to handle it according to extensions, and how we add the *MPL-Option* to such a header the right way. A standard IPv6 header has a minimal length of 40 bytes. It contains the basic information needed to successfully deliver a packet, like a source address, a destination address, payload length and other important parameters. This additional *MPL-Option* extends the basic header by adding a further IPv6-options, which is necessary to route the data message. MPL needs such an additional option, called the *MPL-Option*, to put all of its routing information into the packet header. These fields are an 8-bit *Sequence* number, some control flags and an optional *Seed-ID* that the nodes use as their specific address. The structure of such an *MPL-Option* is shown in Figure 3.2.

The actual length of the *MPL-Option* depends on the used length of the *Seed-ID*. The value of the *S-flags* gives the used length for the *Seed-ID*, whose value 0 indicates that the IPv6 address is the *Seed-ID*, 1 specifies a 16-bit *Seed-ID* identifier, 2 an identifier with 64 bits lengths and an identifier of 3 indicates a length of 132 bits. Every message also needs the *Sequence* field, as every message of a certain *Seed-ID* has a sequence counter, which increments with each new message. Further, there is the *M-flag*, which indicates

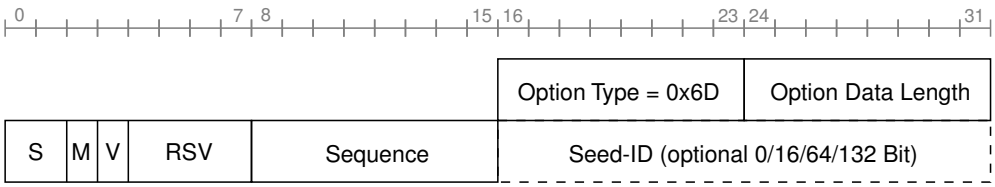


Figure 3.2: Format of an *MPL-Option* used to route data messages [HK16]

that packets *Sequence* number is the highest one known to the sending node for this specific *Seed-ID*. It allows to detect inconsistencies and raise the according *Trickle* event to trigger the process to correct them. Additionally, there are also the *V-Flag* and *RSV* parts of the header. The *V-Flag* is generally always set to 0, as it indicates if the packet conforms to the RFC specification. If this bit is one, a standard MPL implementation has to drop the packet. *RSV* marks some bits that are currently unused and reserved for later use. These bits must be set to 0 and ignored by a receiving node. To form a correct MPL data message, this *MPL-Option* is directly added to the message's IPv6 header, into the so called *Hop-by-Hop* Options Header, which is then followed by the IPv6 protocol's payload.

Control Messages

Next, we will look at the second message type that MPL defines, the *Control Messages*, how and why we need and use them, which data they contain and the situations when we send such a message.

In general, such a *Control Message* contains an overview of all the received *Data Messages* of a node that are not old enough to be deleted. A node can delete data messages to reduce its amount of used memory only if a message's sequence number is lower than a certain minimum sequence number. In this case, the message is considered as old and successfully handled, which allows the node to delete this message if it needs to. Additionally, a node also deletes all information it has from a node due to the expiration of the *SEED_SET_ENTRY_LIFETIME* timeout if it did not receive any messages from it during this period.

Each control-message is a special ICMPv6 message, that is sent to the *control-domain* of an according *data-domain*. The general management and reserved address ranges for these domains are described in [HK16] and [Hab02]. These RFCs define a permanent multicast address for the usage with MPL, the *ALL_MPL_FORWARDERS* multicast address 'FF0X::FC'. The 'X' is a placeholder for the used *multicast address scope* of the sent messages.

MPL specifies, that there are two separate MPL-domains necessary to use both data and control messages, as each domain is only allowed to handle a single message type. The

two IPv6 addresses necessary for these domains only differ by their *multicast address scope* value. The data-domain typically uses a scope value of 3, which forms the address 'FF**03**::FC', while the control domain then uses the Internet Protocol (IP) address 'FF**02**::FC'.

A control-message consists of the basic ICMPv6 fields *Type* (=159), *Code* (= 0) and followed by the standard *ICMP-checksum*. As additional part of their header, these messages have an array of *MPL Seed Info* objects. Figure 3.3 shows the content of such an object and the overall construction of an MPL control-message. Such an *MPL Seed Info* object encodes all the important information that a node currently has on another node's received messages.

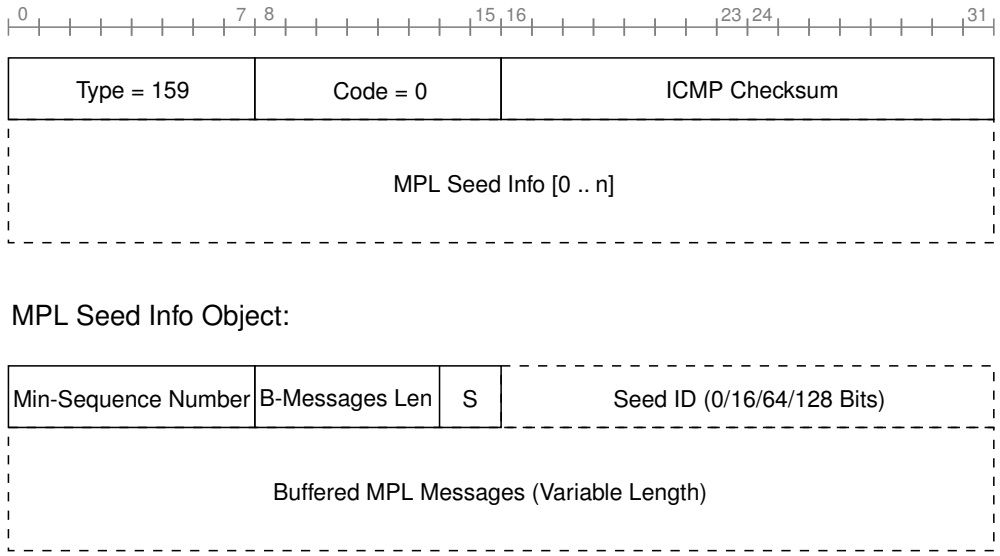


Figure 3.3: The *MPL Control Message* used by the *reactive* operating mode of MPL [HK16]

It contains the currently used *minimal sequence number* for a particular node indicated by the field *Seed ID*. The *S*-field indicates the length of the *Seed-ID* field congruent with the data messages. The *B-Messages Len* field states the length of the *Buffered MPL Messages* data field. This field is a bit-vector, where each set bit indicates that the sending node knows the message. The translation of the bit-index to the actual message's sequence number is calculated by $SeqNo = i + MinSeqNo$, where i is the index of the currently handled bit. A receiving node can decode the content and compare it with its own known message sets to detect inconsistencies and initiate the corresponding steps to correct them.

3.2.4 MPL Operation Modes

Proactive MPL

The *proactive* operation mode of MPL utilizes the *Trickle* algorithm [LCH+11], to schedule and transmit data messages to all the neighbour nodes. Upon reception of a new MPL message, each of these neighbour nodes will store this new message in their specific history of received messages and create and start an associated *Trickle Timer* for this message. If this timer expires and the node has received enough *consistent* trickle events, the node does not retransmit the message. If there were too few *consistent* events, the node retransmits the associated message. In the context of the *proactive* operation mode, such a *consistent* event is the reception of the same data message from different nodes. When there is an inconsistency within the MPL header information (e.g. mismatch of the header flags and sequence number) of a previously received message, this data-inconsistency triggers an *inconsistent* trickle event, and the message's timer is reset (subsection 3.2.1).

Another property of this operation mode is that it sends data messages without additional control messages. However, the nodes only know for themselves which messages they previously received. They have no information and possibility to identify if they received all the messages sent by their neighbour nodes.

This property addresses the first problem of the *proactive* mode. With this mode alone, there is no possibility to check if all the current neighbours received the previous transmissions. All that the *proactive* mode does is automatically retransmit a newly received packet after some time if it did not receive it enough times from its neighbour nodes. Therefore, the reliability improves only by retransmitting the message multiple times. However, there is no acknowledgement or feedback from other nodes if they received the message or if the node missed out on some further data messages.

Therefore, an energy-efficient WSN that uses MPL's *proactive* operation mode needs the following six parameters optimally configured. Otherwise, the network will lack performance and reliability, or its battery will be drained soon due to the high amount of retransmissions. Three out of these six parameters are the ones needed to configure the utilized *Trickle* timers for the transmission of MPL data messages. The fourth one enables the *proactive* operation mode, and the algorithm uses the fifth parameter to specify the maximum number of *Trickle* timer expirations for the transmission of a data message. Therefore, this parameter defines the maximum number of data message retransmissions for this node if there are no *inconsistent* events raised. The last parameter is a watchdog timeout, which is necessary to free old data that is no longer needed. Such old and remaining data are the different messages, trickle timers, sequence-counters, and other temporal data of a node from which we did not receive new messages.

- *PROACTIVE_FORWARDING* [*bool*]
  This parameter simply defines if the *proactive* mode shall be used or not, when a MPL data message is received. If it is deactivated, the routing protocol will not forward any received messages on its own.

- *DATA_MESSAGE_IMIN* [*time*]
  This is the $I_{\min}$ parameter of the *Trickle* algorithm, that acts as the lower bound for the timers initial value and defines the lowest possible delay that can be used for the initial interval of the *Trickle Timer*.

- *DATA_MESSAGE_IMAX* [*int*]
  This integer parameter specifies the value used to calculate the upper border for the *Trickle* algorithm. This upper border is calculated with

$$t_{\max} = 2^{I_{\max}} * I_{\min} \tag{3.2}$$

  defines the longest possible *Trickle* interval, after which the timer runs off and a data message is sent if the *Trickle Timer* fires.

- *DATA_MESSAGE_K* [*int*]
  This parameter declares how many *consistent* events have to be received to inhibit the transmission of a data message when the *Trickle* timer has expired.

- *DATA_MESSAGE_TIMER_EXPIRATIONS* [*int*]
  This parameter indicates the number of *Trickle Timer* expirations are allowed to happen until the timer is deactivated and no further message transmissions are triggered anymore. Therefore, this parameter is used to set the maximum number of initially planned data message transmissions. Although, a the reception of an *inconsistent* event resets the corresponding expirations counter to zero and restarts the *Trickle Timer*.

- *SEED_SET_ENTRY_LIFETIME* [*time*]
  Specifies the idle time after which a node deletes all the information it has of another node. This ensures, that allocated memory freed again, as the node is considered as disabled, if we did not receive any data from it for a this *SEED_SET_ENTRY_LIFETIME* period of time. This parameter is commonly specified in minutes or hours depending on the application's expected idle times.

With these five parameters, we can configure the whole behaviour of the *proactive* mode. In fact, it is possible to configure MPL to behave like the *route-over Flooding* protocol. The two parameters *DATA_MESSAGE_IMIN* and *DATA_MESSAGE_IMAX* provide us with a random *Mesh-Jitter* delay that has to pass before we can send. If we set *DATA_MESSAGE_K* to zero and *DATA_MESSAGE_TIMER_EXPIRATIONS* to one, this will cause the *Trickle Timer* to send the message exactly once, independent of how many *consistent* events we received while the timer was running. The only difference to the *Flooding* protocol is the handling of possible *inconsistent* events that may occur due to differing values for a packets *sequence number* and *M*-flag compared to a nodes local information base. Such a case would trigger a additional sends of the associated message, which is a difference to the *Flooding* protocol.

Reactive MPL

Like with the *proactive*, the *reactive* mode can be completely deactivated, used on its own or together with the *proactive* mode to enable the user all kinds of configurations and combinations of the MPL functionalities. The *reactive* mode uses a much more sophisticated mechanism to achieve stable and reliable transmissions inside a WSN. Instead of simply retransmitting the messages a certain number of times, the *reactive* mode works with *MPL Control messages*. These are ICMPv6 messages introduced in section 3.2.3, that are necessary to compare all the currently known data messages of a single multicast domain. A node encodes all of its currently known data messages from each *Seed-ID*. The node then sends these control messages to all the neighbours to synchronize their data message history with the received one from the neighbour node. Suppose the receiving node detects no differences between the two histories. In that case, a *consistent* event to the control message *Trickle Timer* will prevent the sending of an additional control message. If the histories differ, the *reactive* mode tries to correct these differences by sending additional control or data messages. These differences in the nodes' known information act as the *inconsistent* events for the *Trickle Timer* controlling the sending of the control messages. We have to take the following two cases into account when handling the *inconsistency* events.

- **Case 1:** The node receiving the control message detects that it knows a data message which is not known to the originator of the control message.
  Correction: The node has to reset and start the *Trickle Timers* for each of the missing data messages. These timers will then trigger the retransmission of their corresponding data message. Additionally, a node also has to handle the raised *inconsistent* event accordingly, so the *Trickle Timer* controlling the control messages has to be reset and started again to ensure shorter synchronization periods between the nodes.

- **Case 2:** The node receiving the control message detects that the sending node knows a message, which itself, the receiving node, currently does not know.
  Correction: The receiving node cannot reset the data message timers as it does not know these messages. For this case, the only possible action sequence is first to handle the *inconsistency* event. Second, reset the *Trickle* algorithm for the control message, and send a control message itself, hoping a node knowing the missing data packets detects the inconsistency and triggers the retransmission of the missing data messages as described in the previous **Case 1**.

This additional operation mode addresses the previously highlighted problem of the *proactive* mode that the nodes' buffered messages do not synchronize with the other nodes. For suitable applications, like software updates or parameter re-configurations of the nodes in a WSN, the transmissions must be reliable, and a best-effort transmission scheme will not be sufficient. Especially, as WSNs are not capable to use TCP as their protocol for the *Transport Layer*. Due to the constrained resources of these networks, the

nodes cannot utilize the reliable but complex TCP protocol to detect missing messages. Apart from the general restriction that TCP does not support *multicast* transmissions. Therefore, it handles all its communication as bidirectional unicast sessions. Instead, the nodes use the much simpler UDP, which does not have any additional mechanisms to improve the transmission reliability.

The parameters to configure the transmission of these control messages are similar to the ones previously described for the data messages. Only the amount of used *Trickle Timers* is different, as the *reactive* mode uses only a single *Trickle Timer*, per control domain. Additionally, each unknown data message a node receives triggers a reset of this *Trickle Timer*. The parameters needed to configure the *reactive* mode are listed below.

- *REACTIVE_FORWARDING* [*bool*]
  This parameter defines if the *reactive* mode shall be used or not. If it is deactivated, the routing protocol will not transmit and handle any control-messages.
- *CONTROL_MESSAGE_IMIN* [*time*]
  This is the $I_{\min}$ parameter of the *Trickle* algorithm, which acts as the lower bound of the timers runtime interval. It defines the lowest possible delay that can be used for the initial interval of the *Trickle Timer*.
- *CONTROL_MESSAGE_IMAX* [*int*]
  This integer parameter specifies the value used to calculate the upper border for the *Trickle* algorithms time interval. This upper border is again calculated with

  $$t_{\max} = 2^{I_{\max}} * I_{\min} \tag{3.3}$$

  and defines the longest possible *Trickle* interval, after which the timer runs off and a control message is sent if the *Trickle Timer* fires.
- *CONTROL_MESSAGE_K* [*int*]
  This parameter declares how many *consistent* events have to be received to skip the transmission of the control message.
- *CONTROL_MESSAGE_TIMER_EXPIRATIONS* [*int*]
  This parameter indicates the number of *Trickle Timer* expirations. After the timer has expired this often, it is stopped, and does not send any further control messages until a timer-reset happens and the timer is restarted. This reset is mostly triggered by receiving a new and unknown data message, which acts as an *inconsistent* event and restarts the *Trickle Timer*.

The introduced *reactive* operation mode also has a drawback, which is the high delay that this operation mode introduces. The delay increases particularly with multi-hop transmissions, as for every hop, the sending of the actual data message is initiated via a control message first. Further, the data-message *Trickle Timer* has to expire before

the message is transmitted. A second problem is the overhead introduced by the control messages, since these messages do not contain any application data and can become large for networks with many different senders.

3.3 AODV via IPv6

The last protocol that is discussed in this thesis is the IPv6 enabled version of the basic AODV protocol. AODV in its basic and only IPv4 compatible version is introduced and defined in [PBRD03], while [PR01] defines the necessary adaptions to use AODV in together with IPv6, which enables it for usage as a routing protocol for WSNs, on-top of the 6LoWPAN adaption layer. In comparison to the previously discussed routing mechanisms of *Flooding* and MPL that are both capable of transmitting multicast messages, AODV only works as a unicast routing protocol. Nevertheless, a node can also handle all multicast messages as many unicast transmissions. Therefore, we will compare the differences in performance and energy efficiency when using AODV in this special manner to special multicast protocols. There are even approaches described in [SRRAM19] to simplify and enhance AODV for the usage as a multicast protocol and optimize it for the usage in IoT and WSN applications. This optimization may be necessary, as AODV was initially developed for the usage in MANETs and *ad-hoc* networks, that use WiFi standard *IEEE 802.11* for their physical layer instead of the completely different *IEEE 802.15.4* standard.

The two most important differences between these two standards, as stated by Ting et al. [TEN+11], are the much higher data rate (>10 Mbit/s) and connection reliability that *IEEE 802.11* offers compared to the < 250 kbit/s of transfer rate that *IEEE 802.15.4* achieves. The second and much more important difference is the energy consumption, as the *WiFi* standard is by far not as energy efficient and suitable for resource-constrained devices as the LR-WPAN standard. Therefore, it may be counterproductive to use a routing protocol, which was developed for an entirely different domain and use case.

AODV's basic working principle to find and determine the route to a certain node within a wireless network [PBRD03], is a practical use of the Breath First Search (BFS) graph-searching algorithm [KMT10], which is further enhanced with some caching methods to reuse previously found routes. The introduction of AODV done by Klein-Berndt depicts its basic functionality and used algorithm [KB01].

Each node tries to generate a routing list to the available nodes in the network. If a node detects that the destination is not a neighbour node, it sends a so-called Route Request (RREQ) packet to all its neighbours to ask them if they know the desired destination address as their neighbour. If the neighbour receiving the RREQ also does not know the destination, it again sends another RREQ to continue the search. This action continues until a route to the searched node is found, or the TTL of the RREQ expires, and the search process aborts. In case a node finally knows the destination, it answers with a Route Reply (RREP) message that the intermediate nodes forward to the node starting

this searching process. Each of the nodes receiving an RREP packet updates its routing table to speed up and skip the search for a suitable route for subsequent messages.

Additionally, there are two further messages commonly used to implement AODV, these are the Route Error (RERR) and *HELLO* messages. It uses these messages to update the routing tables of neighbour nodes to populate changes in the network topology and availability of previously reachable nodes. Such a *HELLO* message periodically informs all of a sending node's neighbours of its availability. This message is not forwarded and only used by immediate neighbours. In comparison, the RERR message informs other nodes that they have to delete a routing table entry from their routing tables due to one of the three reasons:

- A node that receives a data packet does not have a routing entry for the packet's destination. Therefore, the node answers with an RERR message to delete the incorrect routing entry in the forwarding node's table.
- If a node receives an RERR message, it has to invalidate and delete a route from its routing table. Additionally, it transmits another subsequent RERR message that contains the deleted nodes, such that this delete instruction propagates through the whole network, and other nodes will also delete these entries from their routing tables.
- When a node detects that the link to one of its neighbours breaks and the neighbour becomes unavailable, this triggers the transmission of a new RERR message to inform other nodes that their routes to this node are not valid anymore.

AODV works quite well with stable and reliable transmissions common in an *ad-hoc* network using the *WiFi* standard. As we will also see in the subsequent chapters, stable and reliable transmissions are essential factors for the performance of AODV. As each falsely received packet triggers RERR messages to delete established routes in all the nodes. This deletion may falsely invalidate routes and cause a new route search only to find the same routing path again. Such a case would create a lot of unnecessary transmissions and overhead in a WSN. As the WSNs generally use *IEEE 802.15.4* as their physical layer, which features very low-power and lossy transmissions, the high number of transmitted messages possibly generates a high amount of transmission failures. These failures again trigger more RREQ and RERR messages, which further increase the message overhead. Another possible problem of AODV together with lossy networks is the frequent sending of *HELLO* messages, which are a special form of RREQs message that is periodically broadcast when the node was idle for some time. These continuous broadcasts permanently consume energy, and in the worst case, a dropped *HELLO* messages can causes a lot of RERR messages, as other nodes have to send RERR messages, if they did not receive a *HELLO* packet from a node for a certain period.

CHAPTER 4

Implementation

This chapter describes the models and components implemented. It contains the description of the new MPL model, LR-WPAN energy model, AODV6 model and the additional *execution and evaluation framework*. These models are developed for the use with NS-3 and can be obtained and used from the public repository of this thesis[1].

4.1 AODV via IPv6

The first new protocol implementation introduced is the simulation model for the AODV6 protocol. This models is not an entirely new simulation model, as there is already a functionally working implementation of AODV available with NS-3. The only problem with this model is its incompatibility for simulations of modern WSNs as it does not support usage with IPv6. Hence, it was only necessary to adapt and port certain parts of the existing AODV protocol's implementation, such that it can be used together with 6LoWPAN and IPv6.

Further, AODV6 is the only protocol used in this thesis that is not capable of transmitting multicast messages. It was nevertheless used to answer and discuss the two following questions and as an entry point to implement NS-3 simulation models.

- Are protocols designed for use with the *IEEE 802.11 (WiFi)* also suitable and reasonable efficient for transmissions via the *IEEE 802.15.4* standard?
- Are multicast protocols even necessary, or is it feasible to utilize unicast messages to transmit the data instead?

To port the existing AODV model to its AODV6 version, the main differences between these two are based on the extended length of the different packet types, due to the

[1] Public thesis-repository: `https://github.com/dlukitsch/ns-3-dev`

increased size of IPv6 addresses, which are now 128-bit long. We have to consider this change of parameter length for all four packet types of AODV6. This increased length of the IP addresses leads to a vast increase in the overall packet sizes as the IP addresses themselves are the main parts of all the four main packet types which are RREQ, RREP, RERR and Route Reply Acknowledgment (RREP-ACK). As AODV6 is completely based on the existing version of AODV, which does not implement the *Flooding* extension description in section 10 of [PR01], this features is also not available in the ported AODV6 version.

Additionally to changing the packet structures, packet handling has to be adapted to accommodate IPv6. Further, the usage of Address Resolution Protocol (ARP) has to be replaced with its pendant Neighbour Discovery Cache for IPv6 (NDISC) and finally the mandatory adjustments for the build systems have to be adjusted.

4.2 Flooding

The second protocol utilized for the simulations in this thesis is the *mesh-under Flooding* multicast routing protocol. It is a simple routing engine, which is commonly used in a broad range of WSN applications. One of these applications is the new *Bluetooth-Mesh* specification [HSPDDCGL+20], which uses this simple and basic *Flooding* approach as their only supported routing protocol.

Due to the importance and simplicity of the *mesh-under Flooding* routing algorithm, this it is available from the NS-3 as part of the 6LoWPAN model. It can instantly be used together with the underlying LR-WPAN model and acts as the most basic entry point to simulate WSNs with NS-3 using its LR-WPAN and 6LoWPAN models.

4.3 MPL

The third routing protocol implemented for the use with NS-3 is MPL. As introduced in section 3.2, this routing protocol relies strongly on the *Trickle* algorithm, which is readily available from the NS-3 *core*-module and acts as an integral part of the MPL implementation.

The class-diagram shown in Figure 4.1, provides an overview of the implemented classes and how they are associated with each other and with the classes of the NS-3 network stack.

To implement a functional routing protocol that can be used together with the IPv6 stack of NS-3, the class implementing the routing logic has to be derived from the abstract class `Ipv6RoutingProtocol`. This inheritance is necessary as the `Ipv6L3Protocol` class requires the implementation of the `Ipv6RoutingProtocol` interface, for any type of IPv6 routing protocol that shall be used with NS-3. `Ipv6L3Protocol` represents the *Layer-3 (Network Layer)* and is responsible for receiving and sending the to the neighbouring layers 2 and 4. To solve the actual routing tasks, it delegates them to its

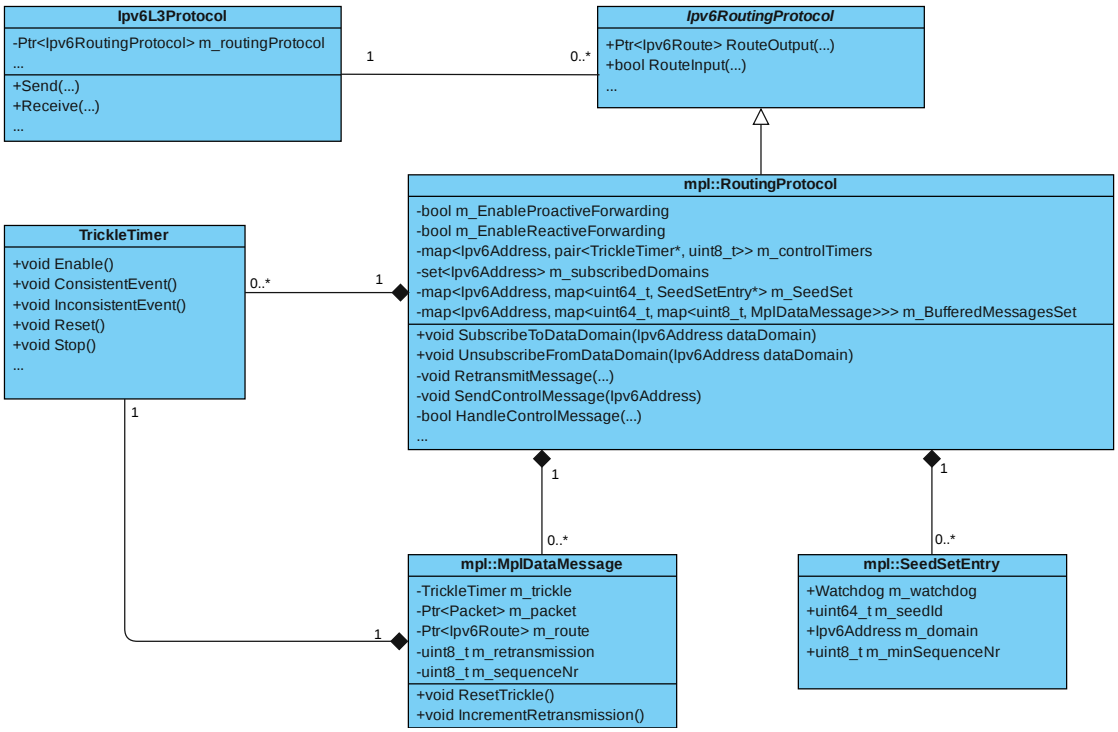


Figure 4.1: Class-diagram of the MPL implementation

linked `Ipv6RoutingProtocol` class-object. Therefore, each `Ipv6RoutingProtocol` has to implement two essential methods, called `RouteOutput()` and `RouteInput()`. `RouteInput()` method implements the algorithms necessary to decide if a message shall be dropped or not. Hence, if an MPL data or control message is received, it removes the MPL specific header parts and uses this information to route the incoming message upwards the protocol stack. Accordingly, the `RouteOutput()` method handles the packets going into the opposite direction and is responsible to route the outgoing message downwards the network stack and add the MPL specific parts of the IPv6 header.

The `mpl::RoutingProtocol` and `Trickle Timer` classes together with the `MplDataMessage` and `SeedSetEntry` classes from the `mpl`-namespace contain the actual implementation of the MPL protocol as specified by [HK16]. This specification defines the different operation modes, parameters, message extensions and the general rules of how to handle the messages. Our implementation follows this specification and also implements the optional memory-management- and housekeeping features to reduce the memory usage as it is necessary for implementations in real resource-constrained nodes. The only two minor restrictions our implementation currently employs is the reduced support of only a single *Seed-ID length* of 16-Bit instead of the different possible lengths specified. The second restriction is the currently missing implementation for *IP-in-IP* tunnelling as specified in [DC98]. We currently omitted these additional functionalities intentionally, as they are not substantially necessary for our scope.

The functionality and details on how MPL works are discussed in section 3.2. Therefore, we now focus on the work necessary to integrate its two operation modes into NS-3 and how to extend the network stack to support the new message types for *MPL-Data-Messages* and *MPL-Control-Messages*.

4.3.1 Proactive MPL

MPL-Option Header

Before we start with the implementation of the *proactive* mode, we first have to extend the IPv6 network stack to support the creation of *MPL-Data-Messages*, that contains an *MPL-Option* within a *Hop-by-Hop Options Header*.

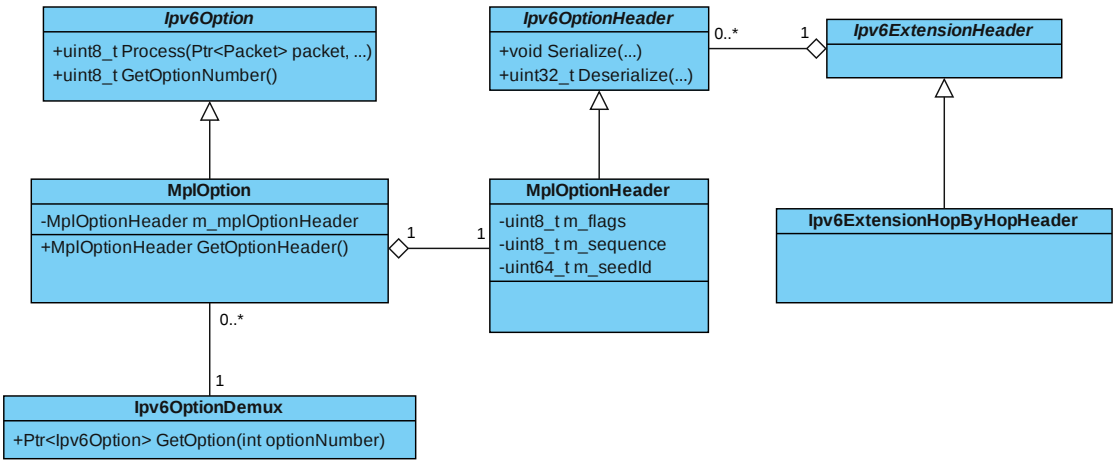


Figure 4.2: Class-diagram of the *MPL-Option* implementation

To implement the *MPL-Option*, we need to add two classes `MplOptionHeader` and `MplOption` as shown in Figure 4.2 to the NS-3-simulator.

The first one, `MplOptionHeader` is the class that contains the relevant parameters that have to be included into an actual *Data-Packet*, while the second class `MplOption` implements the `Process()` method, which is necessary to read a `MplOptionHeader` from a received data-packet. To simplify this complex extraction process, NS-3 provides us with the `Ipv6OptionDemux` class, which allows to easily extract any available `Ipv6OptionHeader` from a data-packet by calling the `GetOptionHeader()` with the header's *OptionType*.

The *reactive* mode is MPL's backbone, as it is responsible for handling the sending, receiving and caching of all the *Data-Messages*. Therefore, the `mpl::RoutingProtocol` objects need three data structures to route *MPL-Data-Messages* for multiple multicast-domains. The following list describes the purpose of each of them and what their intended usage is.

- `set<Ipv6Address> m_subscribedDomains`
  Any node can dynamically join and leave different multicast domains while running and the node needs this data set to keep track of the domains it is currently registered. Every message with a destination address not inside this set is irrelevant and dropped, as the node is currently not subscribed to this domain.

- `map<Ipv6Address, map<uint64_t, SeedSetEntry*> m_SeedSet`
  This two-dimensional map functions as the node's memory to keep track of all the active *Seed-Nodes* within the multicast domain. A *Seed-Node* is the node that generated the message and initially sent it to its neighbour nodes. Upon sending, these nodes insert their *Seed-ID* into the *MPL-Option* of the packet header, which we use to identify the origin of a particular message.

- `map<Ipv6Address, map<uint64_t, map<uint8_t, MplDataMessage* >>> m_BufferedMessageSet`
  It functions as a message cache of variable length containing all the messages that the protocol considers relevant and new enough. Therefore, this *Message Buffer* is implemented as a three-dimensional map, such that we can uniquely acquire a single *Data-Message*, sent from a certain *Seed-ID* to one of a node's subscribed domains.

These data structures function as a node's memory to handle all the routing tasks of MPL. Primarily, the `m_BufferedMessageSet` functions like a large message cache that allows storing data messages and its associated *Trickle Timer*. Most of the further steps necessary to implement the *proactive* mode are functions that manage the insertion, searching and deletion of data for the three data structures. In contrast, the actual sending process of the data messages is solely triggered by the callback function of a data message's *Trickle Timer*. Figure 4.3 shows an activity diagram that depicts the different steps implemented to send and receive a single data message with MPL's *proactive* operation mode.

The sending and receiving processes only differ in a single additional step when receiving a data message, which is necessary to route the packet to the next higher level of the network stack. The remaining steps are similar between sending and receiving, as every received message is forwarded by the node, which uses the same action sequence as a newly sent packet.

The two triggers that initiate a send or receive routing process are the two functions `RouteOutput()` and `RouteInput()`. When calling them with the currently assembled data packet, MPL will first perform the check if the node is subscribed to the MPL domain of the currently handled packet's domain. It performs this check by searching the destination address of the packet within the `m_subscribedDomains` set. The packet is further processed if the right domain is found within this set. Otherwise it is dropped, and the routing finishes. When processing the packet, the next step is to search the node's correct network interface, which has to send/forward the handled packet. If

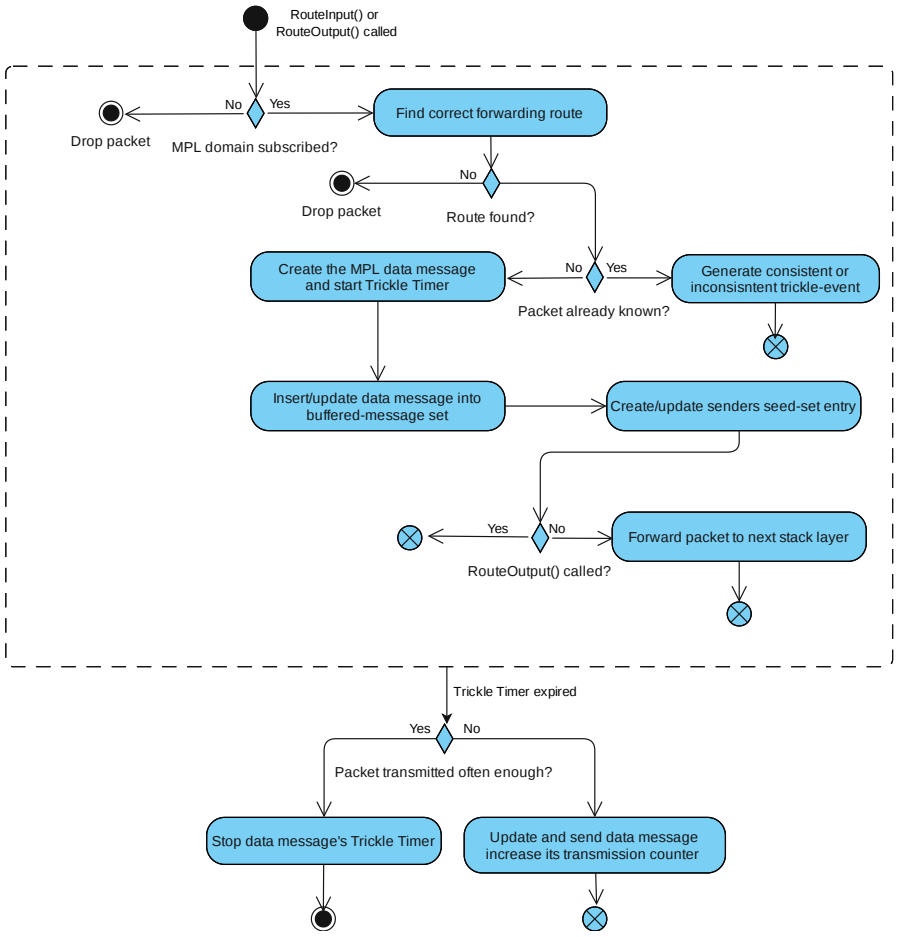


Figure 4.3: Activity diagram to send/receive a data message with the *proactive* mode

such a network interface cannot be found, the routing protocol must again drop the packet as it cannot route it correctly. If there is a suitable route found to send the packet, the following step is to check if the node already knows this particular message or received it for the first time. Therefore, the protocol checks `m_BufferedMessageSet` if it contains an entry for this message. If it does not know the handled message, it has to perform the necessary tasks to create a new data message entry and its associated *Trickle Timer* and to start this new timer. Further, the routing protocol also has to insert the message into the `m_BufferedMessageSet` and update the packet originator's entry in the `m_SeedSet` data structure. If the node knows the handled packet, there exists already a *Trickle Timer* instance for this packet. Thus, it searches the messages and timers in the `m_BufferedMessageSet` and raises either a consistent or inconsistent event. The event type depends on comparing the flags and sequence numbers of the handled and stored packets. If there were any inconsistencies, an inconsistent *Trickle* event is raised, else a consistent one. After these tasks, the only missing one is to forward the packet upwards to the next layer of the network stack if the protocol handles a newly

received data packet.

It is crucial to notice that MPL according to its specification, does not instantly send or forward a packet to its neighbours. Instead, all the actual sending tasks are triggered sometime later by the message's assigned *Trickle Timer*. If this timer fires, it executes the function necessary to send a packet. Therefore, each time a data message's *Trickle Timer* fires, it first has to check if the packet's re-transmission counter is less than *DATA_MESSAGE_TIMER_EXPIRATIONS*, to check if the message must not be sent again. If the message is sent, its values for the re-transmission counter and *M-flag* are updated, and it is passed to the next lower level of the network stack. If the message was already re-transmitted *DATA_MESSAGE_TIMER_EXPIRATIONS* times due to previous timer expirations, the associated Trickle Timer is stopped. When the associated *Trickle Timer* is stopped, it does not trigger any new transmissions for this message until it may be restarted by an *inconsistent* event or the *reactive* mode.

4.3.2 Reactive MPL

The main purpose of the reactive mode is to synchronize the known data messages among neighbour nodes. Therefore, it uses a second message type called control messages and an additional *Trickle Timer* per subscribed multicast domain. For these timers, we use an additional map object called `m_controlTimers`, which uses the domain's IP address as its key and a pointer to the associated *Trickle Timer* object as its value. Like for the *proactive* mode, these additional timers trigger the sending process of the control messages.

Within these control messages, the *reactive* mode encodes all the currently known data messages for a specific multicast domain and *Seed-ID* into several *MPL Seed Info* objects. Each of these *Seed Info* objects consists of a *Min-Sequence Number* and a bit vector, where the *Min-Sequence Number* functions as the offset and indicates which sequence number the bit on position zero stands for. These two data fields, together with the bit-position inside the bit vector, allow us to encode if a data message with a certain sequence number is known, the bit is set, or if not, then the current bit is reset.

These control messages are then sent to the neighbour nodes, which upon reception, compare the *Seed Info* objects from the control message with their currently known messages for the given MPL domain. Therefore, they check if they have the same messages in their `m_BufferedMessageSet` data structures as the node sending the control message. These inconsistencies then raise the according *Trickle Timer* events as described in section 3.2.4 with the aim to correct these differences.

In Figure 4.4, the necessary logic to implement the *reactive* mode are depicted. Just as for the *proactive* mode, a node first has to subscribe to a certain MPL domain, by inserting the IP address of the subscribed domain into `m_subscribedDataDomains` and inserting the IP address, together with the *Trickle Timer* and a counting variable, used to count the timer expirations, into the `m_controlTimers` map object of type

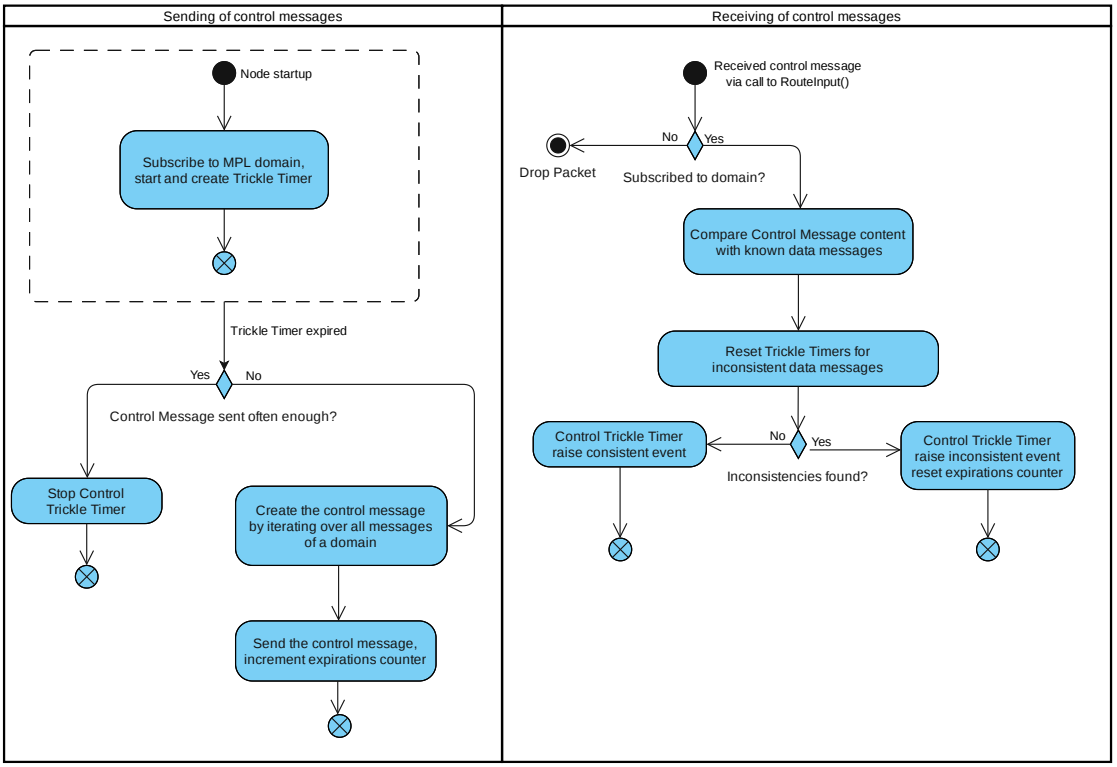


Figure 4.4: Activity diagram to send and receive a control message with the *reactive* mode

`map<Ipv6Address, pair<TrickleTimer*, uint8_t>`. After that, the *reactive* mode is ready and waiting to either send or receive control messages.

To send a control message, the *Trickle Timer* for the control messages has to fire, then the callback function to send a control message is executed, which first checks if there were already `CONTROL_MESSAGE_TIMER_EXPIRATIONS` or not. If there are already that many control messages sent, the timer is stopped, and no further control message is generated until an inconsistent event is detected. Otherwise, we can still send further control messages, the *SendControlMessage()* method iterates over all the *Seed-IDs* of a domain, creates the *Seed Info* objects and inserts them into the new control message it creates. After this step has finished, the *Trickle Timers* expiration counter is incremented by one and the control message is sent via the Internet Control Message Protocol (ICMP) implementation of NS-3.

On the right side of Figure 4.4, we can see the handling of a received control message. Compared to handling data messages, the handling of received control messages is much simpler, as a received node does not forward these messages. The node checks if it subscribed to the domain of the control message and in case the node is subscribed to the domain, it consumes the data and compares the content of the control message with its

own data messages stored in `m_BufferedMessageSet`. Depending on the differences detected, the implementation then raises the according *Trickle* events for its data message *Trickle Timers*.

4.4 LR-WPAN Energy Model

As this thesis aims to evaluate the energy consumption and efficiency of different routing protocols using LR-WPAN and 6LoWPAN, it was necessary to extend the LR-WPAN model with an optional *Energy* model. This additional model is necessary to enable the implementation of the LR-WPAN PHY to simulate its energy consumption. As the *WiFi* model already supports such a model, it was adapted for the use with the intended LR-WPAN model. Hence, the existing LR-WPAN model of NS-3 is extended with its own energy model, that allows to simulate nodes and their connected energy sources. This new model is derived from the existing `wifi-radio-energy-model` [WNP11], and enables the LR-WPAN model to connect battery-objects with its LR-WPAN nodes. Figure 4.5 shows the energy model's Unified Modeling Language (UML) class diagram for the initial implementation of the *WiFi* energy model and the overall *Energy* model framework of NS-3.

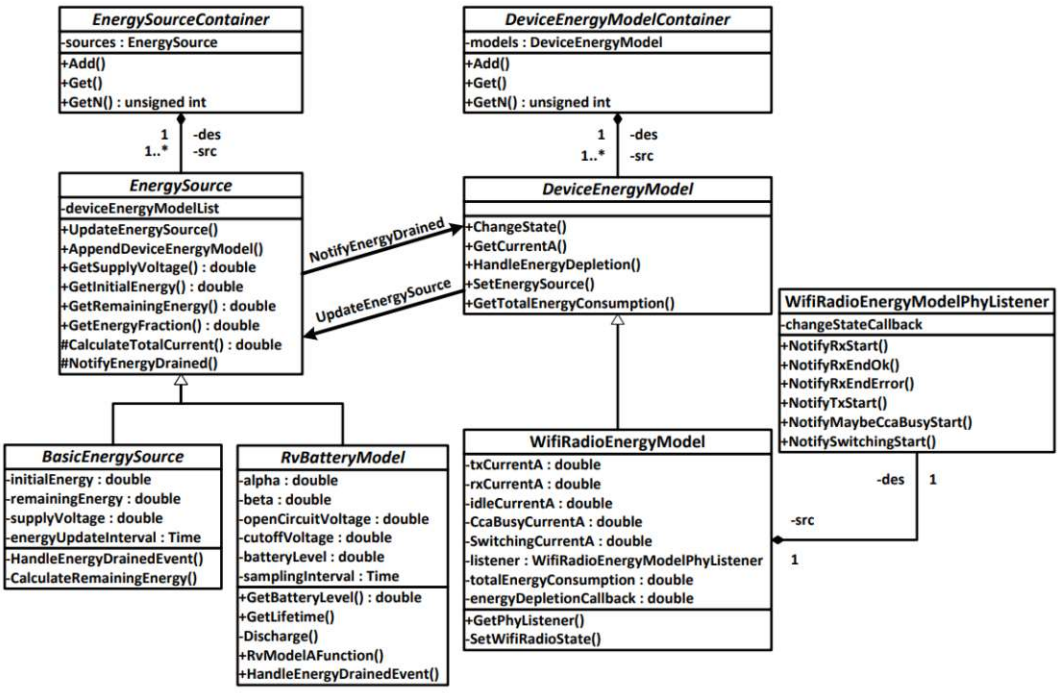


Figure 4.5: UML class diagram showing the NS-3 *WiFi* energy model [WNP11]

This initial model depicting the implementation in the *WiFi* model was ported for the use with NS-3's LR-WPAN model. In order to calculate the virtual energy consumption

of a node, the model is installed onto a node and connects to the nodes' LR-WPAN PHY and monitors its internal states for the sender and receiver. This `lr-wpan-phy` component then notifies the energy model on every state change it performs. The model then uses the information of the states and their active periods to calculate the consumed energy of a node, by multiplying the time period with the drawn current and supply voltage for a specific state. These time-boxed slots are then summed and to calculate the overall energy consumption of a node during the simulation.

With this adaption to NS-3's existing *WiFi* energy model, it is now possible to perform simulations of the nodes' energy consumption within a WSN using LR-WPAN. Additionally, this model allows us to simulate the behaviour of a drained battery that switches off a node when its available energy is drained. A node with such an empty battery is then automatically disabled by turning off its PHY. This is done by switching off both of its internal sender and receiver units leading to a node that does not send and receive any further packets. The node stays completely inactive until the simulation has ended or its battery is recharged again. In case of such a recharge, the node has new energy available, gets functional again and starts to communicate with the other nodes in the network again.

With these implemented features, the model allows us to simulate the detailed lifetime of every node within a WSN. This additionally available data allows us to detect areas inside a network that have a higher energy consumption than others. These additional parameters are tracked with two `TracedValues`, for the *total consumed energy* and the *depletion state* of a node's battery. These *TracedValues* generate a notification each time they are changed and allows us to extract the exact results from the simulations.

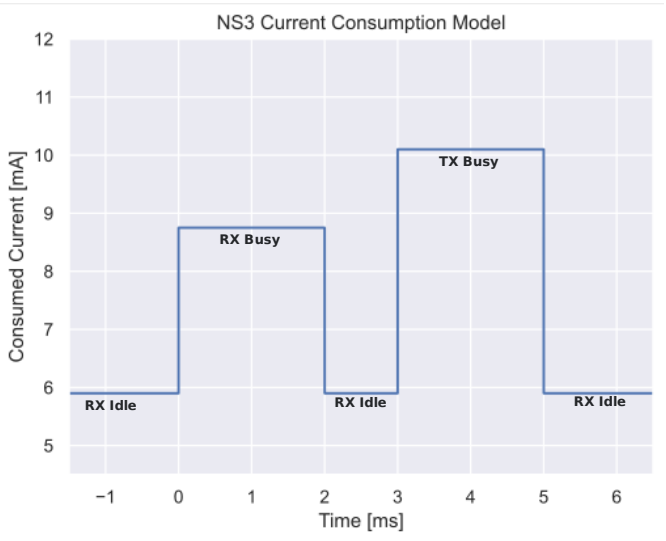


Figure 4.6: States of the NS-3 *current* model to calculate the energy consumption

The model utilizes a similar *TxCurrentModel* as the *WiFi* model, called the `LinearLrWpanTxCurrentModel`. This class is again adapted from the already existing

transmit current model of the *WiFi* component and models the sending current based on the configured gain of the transmitter. The different values configured for the receiver and sender current levels are used from the measurements made by S. Adelmann in [Ade21], who performed a detailed analysis of the sending and receiving processes of the NRF52840 board. Based on these results, we defined the default parameters as stated in Table 4.1. As seen in Figure 4.6, the currently implemented *Current* model is a very simplistic one and models every supported types of the LR-WPAN PHY's current consumption states as rectangles and does not allow to used different types of curves. This limitation on the one hand comes from the monitoring of LR-WPAN PHY's different states. As the current implementation uses only a minimal set of four different states, these state changes are the only information available to the *Energy* model to simulate the PHY's energy consumption. These states provided by the PHY are:

- Receiver Idle
- Receiver Busy
- Sender Busy
- Sender and Receiver Off

To adapt and solve this problem, it might be possible to divide these four states provided by the LR-WPAN PHY into additional sub-states, that can be designed based onto the actually measured energy consumption of a node. Further, the current LR-WPAN PHY's implementation does neither support Radio Duty Cycling (RDC) nor any other method to disable the receiver during idle times, which has a big influence on the overall results and functionality of the new energy model for LR-WPAN. However, improving these limitations would require a complete refactoring and reimplementation of the current PHY, which is completely out of scope and deferred to further work.

The `LinearLrWpanTxCurrentModel` model can be used with the energy model to calculate the current energy consumption based on its dynamic configuration. Since its *Tx Send Current* is not only fixed value, this sending current becomes a linear function which depends on the currently configured sending power given in [dBm]. This allows us to change different properties during simulation to automatically calculate parameters relevant for the energy consumption. The function implemented by the `LinearLrWpanTxCurrentModel` is shown in Equation 4.1. Therefore, if we utilize the default parameters of $V_{bat} = 3V, \eta = 0.1, I_{idle} = 6.74mA$, we acquire a sending current of $I_{Tx} = 10.08mA$ for a sending power of $P_{Send} = 0dBm$, which is equivalent to the measured sending current for the NRF52840 board at +4dBm [Ade21].

$$I_{Tx} = \frac{10^{(P_{Send}-30)*0.1}}{V_{bat} * \eta} + I_{Idle} \tag{4.1}$$

Parameter Name	Value [mA]
Tx Idle Current	5.9
Rx Idle Current	5.9
Tx Send Current	10.1
Rx Receive Current	8.75

Table 4.1: Current values for the LR-WPAN energy model at 0dBm sending power

It is obvious apparent that this linear model, together with the calculation of the energy consumption via the constant current values for the sender and receiver's idle states, is a significant simplification compared to the actual processes happening in real hardware nodes. However, due to the limitation of the states provided by the `lr-wpan-phy` model and given that the energy consumption of a node is very hardware dependent, these formulas are detailed enough to perform meaningful simulations with them.

In Listing 4.1, the example code of how to use this *energy model* in a NS-3 simulation is shown. This code uses the provided *helper-objects* to install an energy source to the object of a physical node, and the energy-model itself gets installed on a certain *NetDevice* of this node. Therefore, this model allows for the utilization of different energy models for every *NetDevice* connected to a node. In contrast, each of these energy models accesses the connected *energy-source* of the node itself.

```
1    BasicEnergySourceHelper basicSourceHelper;
2    EnergySourceContainer sources = basicSourceHelper.Install(nodeContainer);
3
4    LrWpanRadioEnergyModelHelper radioEnergyHelper;
5    DeviceEnergyModelContainer deviceModels = radioEnergyHelper.Install(
     netDeviceContainer, sources);
```

Listing 4.1: Example usage of the energy model for LR-WPAN

4.5 Execution and Evaluation Framework

As we are using NS-3 to perform a large number of simulations over various parameters and configurations, some features of NS-3 to implement ourselves a *framework* aside from the basic NS-3 simulator. As shown in Figure 4.7, this framework allows us to generate simulation topologies quickly, define various parameter configurations for these topologies and automatically use them to configure the NS-3 simulator to perform and evaluate the executed simulations for several different parameters, which it monitors during the simulation.

This section introduces all these implemented functionalities, which allow us to perform automated while reducing configuration overhead. Another positive side effect of these predefined simulation configurations is the reduction of falsely configured simulations,

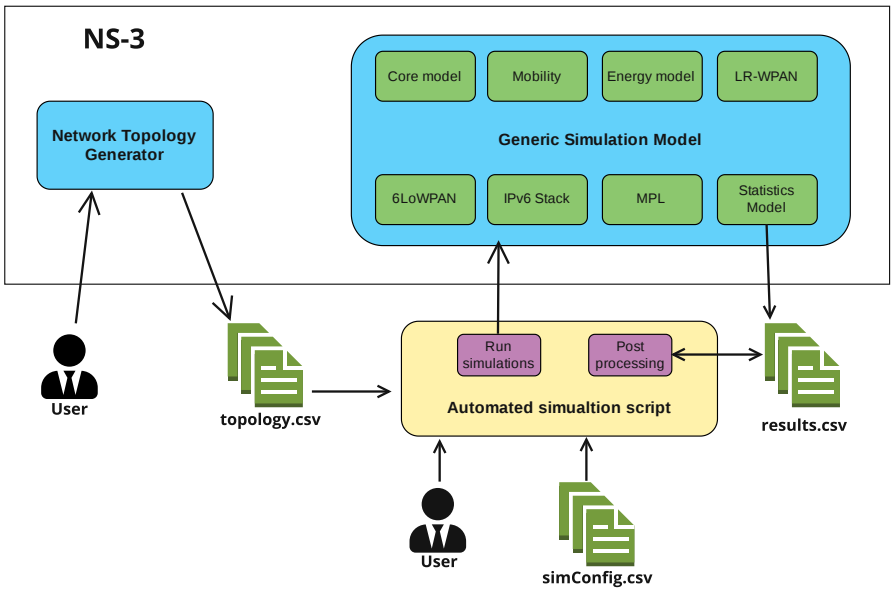


Figure 4.7: Schematic functionality of the *Execution and Evaluation Framework*

as it solves the task of constantly editing simulation parameters within the simulations source code.

4.5.1 Network Topology Generator

The functionality necessary to work with this simulation framework in order to, later on, perform simulations is to create the overall structure and starting positions of the nodes inside a WSN. These generated starting positions for the nodes inside a network are then later on used to create the WSN within NS-3 and position the nodes according to the generated network structure.

A *NetworkGenerator* simulation was implemented for NS-3 that generates and exports different network topology setups to a file. This simulation model is completely parametrised within its source code and uses the different `PositionAllocators` classes provided by NS-3 to generate network topologies. These topologies are encoded in Comma Separated Values (CSV) files. Such a file contains the number of nodes inside the network and initial positions on the grid. In Table 4.2, the typical content and layout of such a generated CSV file is shown, it describes a network consisting of three nodes that are arranged in a line structure and do not have any interference generating nodes.

As the simulation only positions the nodes, the original output file does not contain any definitions of the *source* and optional *sink* nodes. These are the nodes that have the actual NS-3 application-layer functions installed that send messages into the network periodically. As we also implement simulations that use AODV6 and *Flooding* to route their unicast-packets, we also need a possibility to specify the *sink* node where the *source* nodes attempt to send their generated packets. Therefore, after generating and

ID	x	y	z	Sender	Receiver	Interference Sender	Interference Receiver
0	0	0	0	2			
1	110	0	0				
2	220	0	0		0		

Table 4.2: Example output of the *NetworkGenerator* model

manually fine-tuning the specifying the *source* and *sink* nodes as well as adding optional *interference* nodes, this current network topology is ready to be used with the *Generic Simulation Model*.

4.5.2 Evaluation and Statistic Model

The *Execution and Evaluation Frameworks'* primary target is to execute simulations, store and post-process the simulated results automatically. Thus, the implementation of the `Statistic` component was necessary to extract all the different simulation results for from the nodes. This `Statistic` model provides us with the functionality of monitoring, gathering, processing and extracting the data generated by different simulations. It collects all the experimental parameters of a node and needs to be installed directly to every node that has to be monitored. After the simulation stops, the model allows storing the observed simulation results. Additionally, this model contains some *Python* scripts that implement the post-processing of generated simulation data as well as automatically plotting diagrams from the results.

The functionality of this model uses the so-called *TraceSources* that are part of the NS-3 core functionality and execute a certain callback function every time the value of such a special *TracedValue* property changes. This callback allows us to continuously track and evaluate the different results for every node inside a simulation. Such monitored properties are, for example, the sent and received packets, the internal states of the PHY and the energy consumption and battery state of a node. The model uses this information and further refines it into more advanced properties that allow us to understand better what is happening inside every node. Such a tracing function can simply be registered as shown in Listing 4.2 this code connects multiple functions to the provided *TracedSources* of the LR-WPAN-PHY implementation.

```
1    void Statistics::InstallTraces(Ptr<LrWpanPhy> phy, bool withEnergyModel)
2    {
3        phy->TraceConnectWithoutContext("TrxStateValue", MakeCallback(&
    Statistics::TransceiverStateTraceSink , this));
4        phy->TraceConnectWithoutContext("PhyTxDrop", MakeCallback(&Statistics
    ::TxEndDropTraceSink, this));
5        phy->TraceConnectWithoutContext("PhyTxEnd", MakeCallback(&Statistics
    ::TxEndTraceSink, this));
6        phy->TraceConnectWithoutContext("PhyRxEnd", MakeCallback(&Statistics
    ::RxEndTraceSink, this));
```

```
7         phy->TraceConnectWithoutContext("PhyRxDrop", MakeCallback(&Statistics
    ::RxEndDropTraceSink, this));
8         if(withEnergyModel)
9         {
10            phy->GetLrWpanRadioEnergyModel()->TraceConnectWithoutContext("
    TotalEnergyDepleted", MakeCallback(&Statistics::NodeEnergyDepleted, this)
    );
11            phy->GetLrWpanRadioEnergyModel()->TraceConnectWithoutContext("
    TotalEnergyConsumption", MakeCallback(&Statistics::EnergyUsage, this));
12        }
13    }
```

Listing 4.2: Installation of the tracing functions of the *statistic* model

An example of such a tracked property is the *TraceSource* provided to monitor all of the successfully received messages. Therefore, the *statistics* model registers a specific callback functions inside of the nodes' LR-WPAN-PHY, which executes every time a packet is successfully received. Next, we can use this behaviour to implement a simple counter that provides the total number of messages received after the simulation ends. Of course, we can freely extend the code executed inside of these callback functions and use them to implement more complex and advanced metrics. Such an advanced metric is, for example, the *number of different data messages*, where each node tries to store every one of its received data messages into a `map`. This map allows a node to distinguish between new and known messages such that it can increase the associated counter for each newly inserted message.

4.5.3 Generic Simulation Setup

Due to the aim of automating the whole execution process of NS-3, it was feasible to implement a highly configurable, generic simulation setup called `Generic_Model`. Its main purpose is to simplify the configuration work necessary create simulations of different network topologies with different protocols, parameter configurations and functionalities (e.g. *Energy* model).

This generic simulation is completely parametrisable via its command line parameters and allows an automated execution of multiple simulations one after another. The implementation of this simulation itself also lives inside the folder of the previously introduced *NetworkGenerator* and its generated network topology definitions.

The *Generic Simulation Setup* allows to implement and maintain a single implementation that can be used to simulate a complete set of differently configured simulations. Hence, it is only necessary to maintain this single implementation, which highly reduces the amount of maintenance needed to synchronize changes for multiple simulation definitions.

Apart from its generic structure and functionality, this simulation provides other essential features. The first one is the feature of performing multiple pseudo-random simulations that automatically run for a specific configuration. Thus, it provides us with a higher amount of simulation results, allowing us to reduce the possible impact on our results

due to outliers occurring only within a single simulation run. Further, the simulation also stores its used random-generator settings and the results, allowing us to re-simulate and acquire the same values as for the previous runs.

In order to implement an automatic execution functionality for NS-3, this *Generic_Model* uses the `CommandLine` feature from the *NS-3-Core*. This `CommandLine` component implements a automated parsing of command line parameters, this functionality allows to pass arguments from the command line into the readily compiled simulation and use them to configure its actual behaviour. The `CommandLine` functionality implements the whole process of parsing and storing the data into a specified local variable and creates a simple way to pass a high number of arguments via the `waf` build system into the simulation program. Together with the *Automatic Simulation Script* (see subsection 4.5.4), the generated network topologies and separately defined simulation configurations, this generic simulation automatically performs a set of simulations without any necessity of additional user interaction.

The *Generic_Model* further provides us with other useful features that are listed below:

- Fully configurable simulation via command-line parameters
- Reproducible simulation results due to pseudo-random simulation runs
  - Multiple different but reproducible simulation runs with the same configuration setup
  - Seed values get stored to re-run a certain simulation
- Support for multiple routing protocols
  - Fully configurable MPL
    - Proactive mode
    - Reactive mode
  - Flooding
- Setup of a second network aimed to create wireless interference
- Loading of pre-configured network structures with an arbitrary amount of nodes
- Usage of the mobility model to simulate static and movable nodes
- Usage of an optional energy model for each *normal* node
- Generation of correctly dissected *PCAP*-files for every node
- Generation of *Netanim*-trace files to visualize the simulation
- Automatic extraction of simulation results into a CSV-file

4.5.4 Automatic Simulation Script

The main target of our custom *Execution and Evaluation Framework* is to execute a configured set of simulations automatically. We have already introduced and implemented a suitable simulation model and a network topology generator. We finally need a third part that combines the different features and handles the possibly necessary recompilation and automatic execution of the created binary program for the current simulation.

This part is handled by a script called `run_simulations.sh` located at the root folder of the NS-3 repository. This script contains the control logic to perform automated simulations. First, it reads the different simulation configurations from a configuration file, automatically configures and executes them and finally post-processes the acquired results from the different runs of a single simulation setup.

For all these steps to happen, it first reads and interprets a passed configuration file line-by-line, parses its content and triggers the NS-3 simulations one after another via the *waf* build system. The read parameter configurations from the configuration file get passed as command-line arguments to the simulation, and the *Generic_Model* uses them to configure and perform the simulation.

Additionally, this automation script also allows us to automatically execute *post-processing* steps to simplify and combine the considerable amount of data generated by the various simulation runs. The most important one of these *post-processing* scripts employed is the `combineCsvFiles.py`, which is part of the implemented *statistics*-model. Utilizing this script in the *post-processing* steps enables us to automatically combine the data generated by the different simulation runs of a simulation. This `combineCsvFiles.py` script calculates the results for the mean and median, and standard deviation from all of the related simulation-run result-files of an actual simulation. Thus, this script enables a fast, simple and automatic processing for the subsequent work relying on these calculated results.

CHAPTER 5

Used Models, Simulation Parameters and Metrics

This chapter introduces the different model variations used to obtain the described results from the subsequent chapter 6. The model definitions are used as the base topologies for the various performed simulations and parameter variations. Further, this chapter introduces the different metrics used to compare the performance of the simulated routing protocols.

As it is simply impossible to consider all the aspects, layouts and functionalities a WSN can have, it is necessary to restrict the simulated use cases to a representative subset. Therefore, specific structures and network sizes are selected, that ensure representative simulations, while also being compact enough to effectively analyse them to draw the resulting conclusions from their results. Additionally, the virtual results can then be compared with the measurements and behaviour obtained from actual hardware nodes, that are described in the thesis of Adelmann [Ade21].

A WSN model has many different properties, affecting the energy efficiency and quality of the performed transmissions. These properties are divided into different parameter subgroups. The first group of properties are every single node's physical and hardware properties and the parameters of the whole WSN. Such parameters are, for example, the number of nodes inside the network, the physical distances between them, the energy available from the nodes' batteries and many other parameters of a single node. The second group to distinguish is the utilized routing protocol together with its configuration. These parameters can be kept constant throughout the whole network, or they can be specialized and optimized for every node to increase the performance and lifetime of the overall network without further maintenance to replace the node's batteries to keep the network functional. At last, there is the group of external and transient parameters that influence each node's transmissions differently. Such external parameters are, for example,

the amount of interference a node has to deal with while sending or receiving a message. The influence of interference is generally completely different throughout the whole network, and it has to be considered when designing a WSN, as it can drastically affect the performance of a resource-constrained network. The actual amount of interference a node has to handle depends on the exact distance to the interference generator, and the time the interference blocks the transmission channel and therefore prohibits a successful transmission.

With the results obtained from the simulations of these different network models the performance of the used routing protocols can be classified and compared. With this found classification configuration to find the best performance in transmission reliability and energy efficiency for a particular use case.

Next, the different network configurations for the simulations performed in this thesis are introduced. Some general properties are kept consistent throughout all the performed simulations. Such properties are the number of different simulation runs for a single model, the size of the data packet generated by the sender nodes and many more. In Table 5.1, all of these fixed properties are listed.

Further, all the used models consist of a single sender node that generates the initial messages, periodically sending a fixed amount of equally formed messages with regular payload size to the nodes inside the network. The nodes handle their received messages according to their activated routing protocols, and forward them to the other nodes inside the network. After the simulation has finished, various properties are collected to calculate metrics from them, such as the number of sent and received packets and how many packets were lost during transmission and did not reach their designated destinations. These metrics are then used to compare the routing protocols and their configurations against each other.

The generated models can be divided into the following three separate model types according to their unique properties. These groups are basic simulations of standard network topologies, the simulations of interference scenarios and energy depletion simulations. These depletion simulations utilize the energy model for LR-WPAN to simulate a node with connected batteries.

5.1 Simulation Models

First, the used model topologies used in the simulations are introduced. These models are used to show the performance of each protocol configuration on static WSN, which are very common for many applications, like monitoring specific agricultural processes or measuring weather conditions. As these applications are critical in the energy-efficient and reliable WSNs and need to work with minimal maintenance, the simulations of these static models provide the data for their real-world counterparts.

The models use a fixed distance between every node inside such a model without the effect of external interference. As the transmission quality declines, the further a node

Parameter	Value
Simulation File	Generic Model
Distance between Nodes	110m
Transmission Success Rate	75%
Simulation Runs	5 Runs
Data Packet Size	20 Bytes
Number of Data Packets	60 Packets
Sending Intervall	60s
Simulation Endtime	3800s
Transmit Power	0dBm
Channel Number	Channel 11
MPL Seed-Lifetime	30min
MPL Data I_{min}	1s
MPL Data I_{max}	3
MPL Data K	1
MPL Control I_{min}	3s
MPL Control I_{max}	3
MPL Control K	1
Flooding Message Cache	10 Messages
Flooding Hops	125
Initial Battery Energy	500J
Interference Send Intervall	5ms
Interference Packet Size	50 Bytes
Interference Transmit Power	4dBm
Interference Channel	Channel 11

Table 5.1: Simulation parameters with constant values throughout all performed simulations

is away from another sending node. It is necessary to determine the distance between two nodes that best serves our purpose to be unreliable and unstable enough while still ensuring that most packets are successfully delivered. Therefore, the Packet Delivery Ratio (PDR) between two nodes for the performed simulations is fixed with a value of approximately 0.75%.

Hence, it is necessary to acquire the physical distance between two nodes that results in a PDR of 0.75%. This task is done with an an additional simulation called `lr-wpan-error-distance-plot`. This simulation of a LR-WPAN model generates the diagram showing the simulated PDR for the corresponding distance between two nodes. Figure 5.1 shows the result of this simulation run. As depicted, the transmissions are quite reliable until the distance between the two nodes becomes greater than $100m$. When further increasing the distance, the PDR drops rapidly on the next $30m$. Therefore, at approximately $130m$ of distance, the nodes have reached zero connectivity and cannot

receive any packets from each other.

The target is to have a PDR of 0.75% for our simulations. This target PDR can be achieved with a gap of 110m between two nodes. This distance is also utilized in the following declarations of the different model topologies and setups.

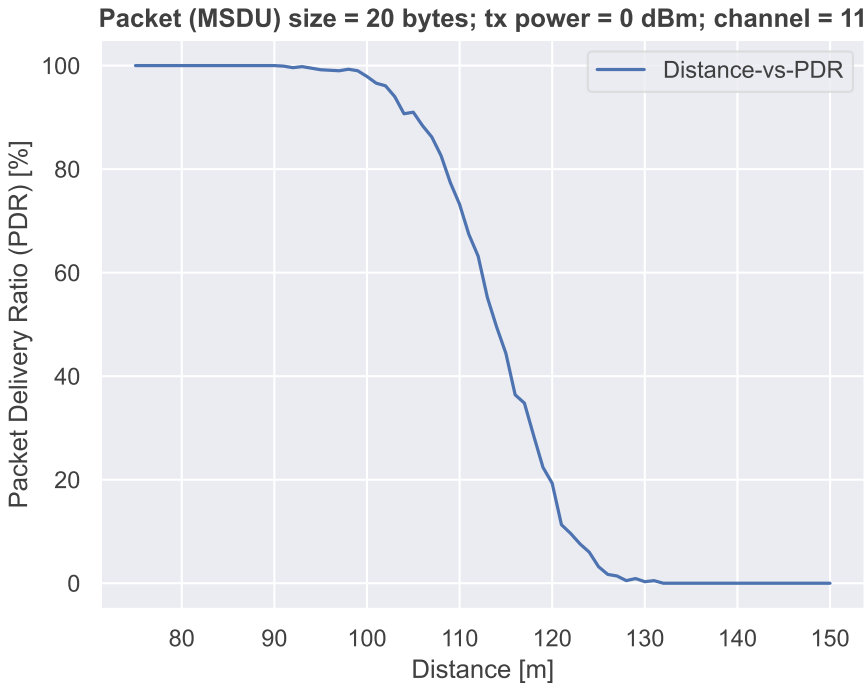


Figure 5.1: PDR of LR-WPAN transmissions in NS-3

5.1.1 Line Structure

The *Line Structure* is probably the most basic structure to build with a certain number of nodes. As its name suggests, it is a WSN that builds a straight line with a constant distance between all nodes, where the sender-node is on one specific end of the line. Although this network topology is quite simple to build and does not look incredibly complicated, it has extraordinary properties and difficulties for unreliable transmission channels. One of these is the absence of redundant transmission paths within the network. Suppose a node wants to send a message from one end to the other, and a single transmission during one of the intermediate node hops fails. That means the intended destination cannot receive the data without an advanced protocol that handles such failed transmissions and tries to resend the packet.

Another exciting aspect of this topology is its assumed energy consumption throughout the different nodes of the network. Assuming that its message traffic originates from the same leaf node of the network, their energy consumption throughout the overall network is likely to be very inhomogeneous. That causes the initial nodes to send and receive

messages more often, depleting energy sources faster. In contrast, the nodes on the opposite side of the network may receive far fewer transmissions, and therefore the energy usage may be significantly less. This behaviour comes from its inherent problem that the Packet Error Ratio (PER) which calculates with $PER = 1 - PDR$, add up for every message hop. So the number of packets that reach the next node decreases significantly after each hop. Due to the previously selected PDR of 0.75%, it is safe to assume that in the case of a *Line* topology nodes which are five or more intermediate hops away from the sender will not receive any messages. The only possibility to improve this number is by implementing additional packet retransmissions that increase the number of hops a packet can sustain.

Of course, such a setup is an entirely constructed case. It would be very irresponsible to design a *Line* topology with such a transmission quality and instead reduce the distance between the nodes to achieve an optimal link quality between two nodes. However, as soon as there is a switch to real-world applications, there may be further effects that influence a networks behaviour, like changes in the environmental temperature and humidity that reduce the sending range of a node, as described by Luomala and Hakala in [LH15]. Due to these effects described by their work, our seemingly constructed use case with this rather low PDR of only 75% is not completely unlikely to occur in a real world environment.

Hence, this seemingly constructed use case can be considered as a possible and plausible use case for a WSN during its overall lifetime. Therefore, our aim is to observe the occurring problems and possibilities and what is necessary to overcome such problems with different routing protocols and their corresponding impact on the energy efficiency of the whole network.

5.1.2 Grid Structure

The grid structure in comparison is a completely different topology than the *Line* setup. Compared with the previously described *Line* structure, the main difference between this topology is the increased number of neighbour nodes as it is a much denser network. With this topology, each node has at least two distinct neighbours if it is a corner node and a maximum of four neighbours if a node is not a border node. This case increases the likelihood of a successful transmission as there are multiple forwarders for a single transmission now. In Figure 5.2, a transmission in such a regular *Grid* topology is shown, where the diagonal nodes 10, 12, 20 and 22 are outside the effective sending range and thus cannot receive the packet. Therefore the datagram can only be delivered vertically and horizontally throughout the whole network. However, compared with the *Line* structure, this topology provides us with much more redundant paths to successfully route packets through the network. This allows even the *Flooding* protocol to quite successfully transmit messages throughout the whole network, which is not really successful for the *Line* topology with similar transmission properties. Generally, the expectation is that a nodes' energy consumption distributes more uniformly throughout the whole network due to these redundant transmission paths. However, there might be

a higher energy consumption for the nodes inside the middle of the grid, as each node has a higher number of neighbour nodes. So these multiple received messages from different neighbours might cause these nodes to receive a increases number of messages, leading to higher energy consumption for these specific nodes.

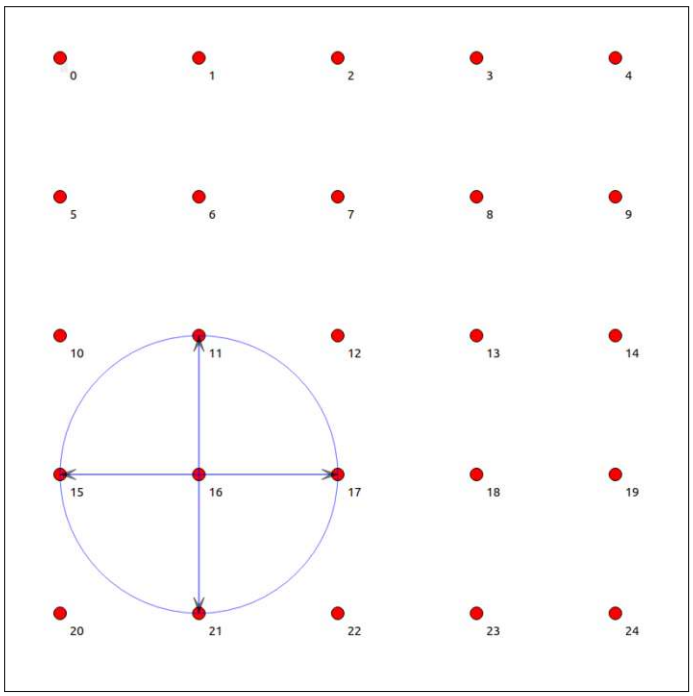


Figure 5.2: Packet transmission in a regular *Grid* topology

5.1.3 Circle Structure

The topology of a *Circle* is an exceptional combination of the *Line* and *Grid* structure, as in a circle, each sending node has precisely two paths to reach a specific destination node within the overall circle. Therefore, compared with the single path of *Line*- and the multiple redundant paths within the *Grid*-structure, there are precisely two redundant paths with the same length and intermediate hops to transmit the packet to the node on the opposite side of the circle. With this model, it can be evaluated how much a second path increases the overall PDR and what this means for the energy consumption of the whole network. In Figure 5.3, such a sending process inside the *Circle* topology and its separate paths to reach node five are shown.

5.2 Special Simulation Models

Additionally to the already introduced simulation models, some specialized simulations are used to determine the behaviour of the WSN and its utilized routing protocol in certain different environment setups that limit the functionality of the nodes. These simulations

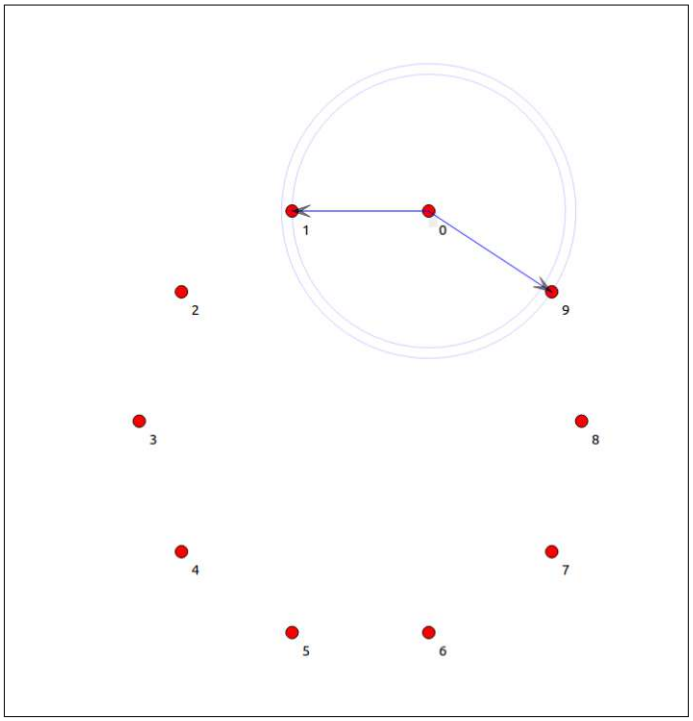


Figure 5.3: Packet transmission showing the redundant paths in a *Circle* topology

provide further data and insight into the robustness against external interference and the same lifetime of a WSN and its nodes. These simulations provide the results that are use to take additional realistic environment aspects into account that cannot be covered with the models from section 5.1.

5.2.1 Network Lifetime Simulations

The *Network Lifetime Simulations* utilize the already introduced simulation topologies. However, compared to the previous simulations, the nodes in these simulations do not have endless energy available. This depletion leads to the dynamic reduction of nodes inside the network as the node shuts down if its battery depletes. So the network's topology dynamically changes depending on each node's actual energy consumption. Further, we cannot only determine a network's change throughout its lifetime but also how long it stays operational and how fast a certain percentage of nodes shuts down and when the network can be considered dysfunctional. Together with the *standard* simulations, which do not consider the nodes' energy depletion but instead measure the overall energy consumption throughout the simulation, these results provide us with the information of which nodes inside a certain topology are drained first and how this depletion affects the performance and energy consumption of the remaining nodes within the WSN.

5.2.2 Interference Simulations

Another specialized simulation type used is the set of *Interference Simulations*. This type of simulation setup extends the existing models with additional *interference nodes*, which are not part of the basic WSN. Instead, such *interference nodes* form a separate network, with the intention to periodically send messages on the same frequency channel as our main network. Depending on how frequently and big the sent interference messages are, they block the wireless medium for a certain amount of time. While this period, the medium is blocked for all the nodes within sending range of the *interference nodes*. This distortion of the medium leads to an increased traffic on the wireless medium, such that our WSN nodes have to deal with this type of disruption accordingly. However, compared to simply reducing the PDR of a single transmission, the simulations provide selective distortions of the wireless medium that a node's routing protocol has to handle accordingly. The *Interference* simulations provide better details on how a specific routing protocol configuration can deal with external influence on the medium, which is a critical property when designing simulations for as realistically environment setup as possible.

5.3 Parameter Variations

Next, the different parameter setups of the routing protocols are introduced. These configurations are utilized to simulate the different WSN topologies and routing protocols. Further, explicit names for these setups are defined that allow a better readability and interpretability of the later discussed results.

Configuration Name	Data $I_{\min}$	Data $I_{\max}$	Data K	Data Expiration's	Control $I_{\min}$	Control $I_{\max}$	Control K	Control Expiration's
MPL 1D0C	1s	3	1	1	—	—	—	—
MPL 1D1C	1s	3	1	1	3s	3	1	1
MPL 2D0C	1s	3	1	2	—	—	—	—
MPL 2D1C	1s	3	1	2	3s	3	1	1
MPL 2D2C	1s	3	1	2	3s	3	1	2
MPL 2D1K2C2K	1s	3	1	2	3s	3	2	2
MPL 2D2K2C1K	1s	3	2	2	3s	3	1	2
MPL 2D2K2C2K	1s	3	2	2	3s	3	2	2

Table 5.2: Utilized MPL parameter configurations and naming scheme

In Table 5.2 the mainly used parameter setups and names of the simulations utilizing MPL are shown, to indicate the used naming scheme. This scheme is used to instantly identify the used simulation parameters for a simulation form its name.

The naming scheme uses $\underline{X}D$ and $\underline{X}C$ as part of its name, to indicate how many data and control messages are configured and if the *reactive* mode is enabled. Further, if the simulation name also contains a $\underline{X}K$, its name indicates the used *redundancy constants* for the *proactive* and *reactive* operation mode. With this naming scheme the MPL configuration and its variable parameters can be directly identified, while all the remaining parameters not contained in the simulation name are kept constant.

Table 5.2 shows some of the later used configurations and how their parameters build up the simulation name. As seen, not every single the parameter that the MPL protocol offers is varied, instead two of the basic *Trickle Timer* parameters ($I_{\min}$, $I_{\max}$) are kept constant, and only change the two *redundancy constants* K and both of the MPL-specific *Expiration* parameters. These *Expiration's* parameters controls how often the a *Trickle Timer* of a *Data* or *Control* message is restarted. Therefore, these parameters define how often a certain message is sent at most.

Another parameter that can be quite tricky to choose correctly are the *redundancy constants* K of the used *Trickle Timers*. These parameters control the so-called '*polite gossip*' mechanism of MPL, by defining how often a message has to be received by a node until it cancels the retransmission of this message for itself. Generally, optimal values for these parameters depend on the network's actual topology and density. Therefore, selecting a K that is too big for a sparse network minimizes the '*polite gossip*' effect. Which causes nodes to not receive the same message often enough to cancel its own retransmission of this message. Therefore, the node will unnecessarily repeat certain messages as their transmissions never get cancelled, which creates energy inefficiencies. Opposed to that, a to low configured value for K used in a dense network may create too many cancelled messages, which may prevent neighbour nodes from receiving particular messages at all, which again can be countered by increasing the number of data message expirations or by enabling the *reactive* mode and use the active synchronization mechanism to improve the performance, but as this mode uses its own K constant it faces the same problem again.

Accordingly, there are three additional MPL configurations where the K parameter for the *data* and *control* messages is changed. With these additional variations of the K parameter, to evaluate the impact on the routing protocol's performance, energy efficiency and which configuration is best suited for each network topology. All these parameter variations and their simulation results, provide data that helps to understand how the changes of a single parameter will likely affect the overall network behaviour.

Due to the limited amount of parameters to configure and the total absence of *control* messages for the *Flooding* protocol, there is only one configuration that is used for the simulation with the *Flooding* protocol. This routing protocol has only three important parameters to configure. The first one is the length of the *Message Cache* that stores several messages our node currently received. The second parameter is the TTL or the number of hops till a node has to drop the packet by force to prevent it from circling uncontrolled through the network. At last, there is the maximum value for the *Mesh Jitter* that randomly delays the sending in different nodes in order to prevent collisions.

As the expectation is that none of these available parameters cause a significant change in the network's behaviour if they somehow suit the deployed network, there are no parameter variations done for the *Flooding* protocol. Therefore, we are explicitly excluding the possible occurrence and causes of a wireless *Broadcast-Storm*, which can occur for totally wrong configured values for the *Message Cache* and TTL parameters. Such a

Broadcast-Storm is considered as an exceptional and disastrous circumstance, which is out of the scope of this thesis.

5.4 Protocol Metrics

Next, the metrics used and defined to compare the simulation results against each other are introduced. The metrics are obtained and derived from the acquired simulation results and try to make the energy consumption of the WSN comparable. To achieve this, the metrics combine specific simulation results aspects and calculate a numeric value, that is then used to compare the different routing protocol configurations and their performances and energy efficiency against each other.

The used metrics are divided into two separate groups, the ones which are completely *hardware independent* and not relying on hardware specific parameters. These metrics do not need fine grained details of a certain node's hardware parameters. Therefore, these metrics enable the general comparison of different routing protocol setups without any a-priory knowledge of the underlying hardware components.

The second group are the hardware specific metrics, the results for these metrics are dependent on certain hardware parameters that have to be configured when simulating the WSNs. As these metric's values are strongly depending on the configured hardware parameters, they can be used to determine and compare the performance of different routing protocols on a specific hardware platform. Additionally, the *hardware-dependent* metrics also allow us to compare a routing protocol's energy efficiency for different hardware platforms to determine the most energy efficient ones.

5.4.1 Hardware-Independent Metrics

As introduces, these metrics do not depend on the hardware specific parameters configured within a certain simulation. Hence, these metrics allow us to compare a WSN and its utilized routing protocol for its energy efficiency, although, these metrics are completely independently from the configured hardware. Due to their independence from the underlying hardware, these cannot provide us with an absolute result and direct comparison for a purely hardware-depending parameter like the energy consumption of a network. Instead, these metrics only allow a indirect comparison, nonetheless, these values are expected to directly correlate with the energy consumption and performance of the network. The *hardware-independent* metrics used are:

- **Number of Sent Messages:**
  This metric counts every message sent for a single node and the whole network. This metric completely ignores if a packet is actually data or controls the message.
- **Number of Received Messages:**
  This metric is similar to the previously introduced one and counts all of the messages received by a specific node. It allows to

- **Transmitted Overhead:**
  This metric calculates how much bytes of overhead a routing protocol sends. Therefore, it provides the ratio between the totally sent bytes and the amount of bytes necessary to deliver every data message exactly once. Therefore, this metric indicates how much additional bytes are sent as overhead within the whole network. The minimal amount of transmitted bytes that does not have any overhead is the case where every unique data message is only sent once by every node within the network.

- **Delivered Data Message Ratio:**
  This metric indicates how many of the unique and initially sent data messages are successfully delivered to all the receiving nodes within a multicast domain. Hence, this metric is the ratio of how many new and previously unknown data messages a node receives. With this metric, we can compare how well the routing protocols perform to deliver a set of messages.

5.4.2 Hardware-Dependent Metrics

The *hardware dependent* metrics are the counterpart to the previously introduced *hardware independent* ones. All of these *hardware dependent* metrics depend on the energy consumption simulated by the LR-WPAN *energy model*. This *energy model* again needs a realistic configuration of the current values drawn by the sender and receiver parts of a hardware node's PHY. The *hardware dependent* metrics use are:

- **Total Energy Consumption**:
  For this metric, we add up all of the nodes consumed energy to determine the network's total energy consumption. This metric allows us to categorize the protocol configurations by their overall energy consumption without considering them with other parameters. This metric applied to a simulation using the LR-WPAN module of NS-3 has the drawback of a very high baseline energy consumption due to the implementation of the MAC component, does not support any form of RDC or similar approach to keep the PHY's receiver disabled for as long as possible.

- **Energy Consumption per Delivered Data Message:**
  Using only the consumed energy of a node on its own may provide misleading results, as it does not consider if the network is still functional and messages get successfully delivered. To solve this problem, we combine the energy consumption of a node or network with its ratio for the *Delivered Data Message Ratio* to compare how energy efficient the different routing protocols work.

- **Network Lifetime**
  The *Network Lifetime* metric defines how long a certain amount of nodes within a WSN stays operational. A network is considered dysfunctional, when a certain amount of nodes depleted their connected batteries' energy and switched off. As

the nodes to not deplete linearly throughout the networks lifetime, this metric is a tuple of three values, where each value uses a different amount of nodes to consider a network dysfunctional. The first value gives the timestamp at with the first node within the network is depleted, the second value indicates when 50% of the networks nodes are inoperable and the third value is the time, where the penultimate node depleted and a communication between two nodes is not possible anymore.

- **Network Lifetime per Delivered Data Messages**
  This metric sets the previously introduced *Network Lifetime* metric in relation to the achieved *Delivered Data Message Ratio*. Therefore, this metric indicates, which network configurations are operable for the longes amount of time, while also achieving the best data message delivery performance. This is necessary as the *Network Lifetime* itself does not guarantee that the network is still capable of successfully sending messages to all the remaining nodes within the network.

CHAPTER 6

Evaluation and Discussion

This chapter contains the evaluation and comparison of the different simulation results, as well as the discussion and our answers to this thesis' initially proposed research questions. Further, all the source code and configuration files necessary to re-perform the deterministic simulations and acquire the same results, as the ones used for the evaluation and discussion, are publicly accessible via the *Github* repository[1] of this thesis.

6.1 Multicast via Unicast Transmissions

The simulations performed first were the ones utilizing the ported AODV6 routing protocol. The aim of these AODV6 simulations is to show that there is a substantial need for maximum energy efficient routing protocols, that cannot be satisfied by re-using existing routing mechanisms that were developed or the use within a completely different wireless network domain, like for example AODV and its origin from MANETs.

Additionally, these simulations and their results show the eminent weaknesses of AODV6, when used for routing protocols in a WSN. One of the main problems of AODV6 is its sheer complexity and a-priori knowledge about a WSN which is necessary to configure the routing protocol correctly. As a quick comparison, the implementation of AODV6 in NS-3 needs a total of 24 different configuration parameters, that have to suit the WSN where the routing protocol is supposed to be used with. In contrast the *Flooding* protocol only needs a set of 4 parameters and MPL only 11, which are divided into two similar sets of 5 parameters used to configure the two operation modes. Hence, this simple comparison of the configuration possibilities shows the first problem when using AODV6 with a WSN, which is its enormous complexity and that it can only be used to transmit unicast messages. Another enormous drawback of the AODV6 protocol

[1] Repository containing the source code to acquire the described simulation results:`https://github.com/dlukitsch/ns-3-dev`

compared to MPL and *Flooding*, is its dependency that the route returned by a successful RREQ stays valid and all the messages get delivered successfully, as AODV6 does only use hop-by-hop unicast messages that are only sent to nodes from the found route, so apart from initially detecting the route, AODV6 does not use the multicast-capability of wireless transmissions to transmit its unicast messages. Therefore, if one of the route's intermediate links drops a message, the sending node reacts by multi casting RERR messages to all its neighbours, which further forward these error-messages throughout the whole network, invalidating the route, such that a new RREQ is necessary to send to the same message again.

Therefore, if a message is dropped when using AODV6, this causes a lot of network traffic that vastly affects the overall energy efficiency of the whole network. If such a routing protocol is then used within a low-power and lossy network like a WSN, where messages are dropped quite frequently, the resulting energy efficiency is simply not suitable for the use within an energy constrained WSN trimmed for maximum node lifetime and transmission performance.

This expected behaviour was verified by simulation results we acquired. As long as the distances between the nodes were small enough, such that all the messages are successfully transmitted and there are no dropped messages, AODV6 works quite well to deliver its unicast packets. Although, if the space between the nodes is increased such that the PDR drops, this causes the routing protocol to constantly invalidate the network routes rendering it unusable to transmit actual data messages. As the AODV6 protocol already struggles to succeed with its main purpose of transmitting unicast messages within a basic WSN to a single receiver node, while at the same time completely draining the nodes' batteries. Hence, the performance of AODV6 for a single periodic unicast transmission is already quite bad and gets even worse when trying to emulate a multicast transmission by sending unicast messages to all of the destination nodes.

These results explicitly show the necessity for more reliable, energy efficient and scalable routing protocols in the domain of WSNs. These protocols have to exploit the intrinsic broadcasting ability of wireless networks to achieve optimal performances and multicast capabilities. Therefore, it is simply unrealistic and unfeasible to re-use a protocol like AODV from the MANET domain whose algorithm is mostly designed on the usage with *WiFi* and expect it to perform well either for unicast or multicast transmissions.

6.2 Basic Model Simulations

To compare the two remaining routing protocols *Flooding* and MPL for their energy consumption and reliability, we performed benchmark simulations with a fixed use case of a single initial sender node. This node sends a new application data packet every minute until it has generated a total of 60 different data messages. These packets shall then be distributed to all the other nodes inside the network. The different network topologies simulated are the *Line*, *Circle* and *Grid* structures introduced in chapter 5.

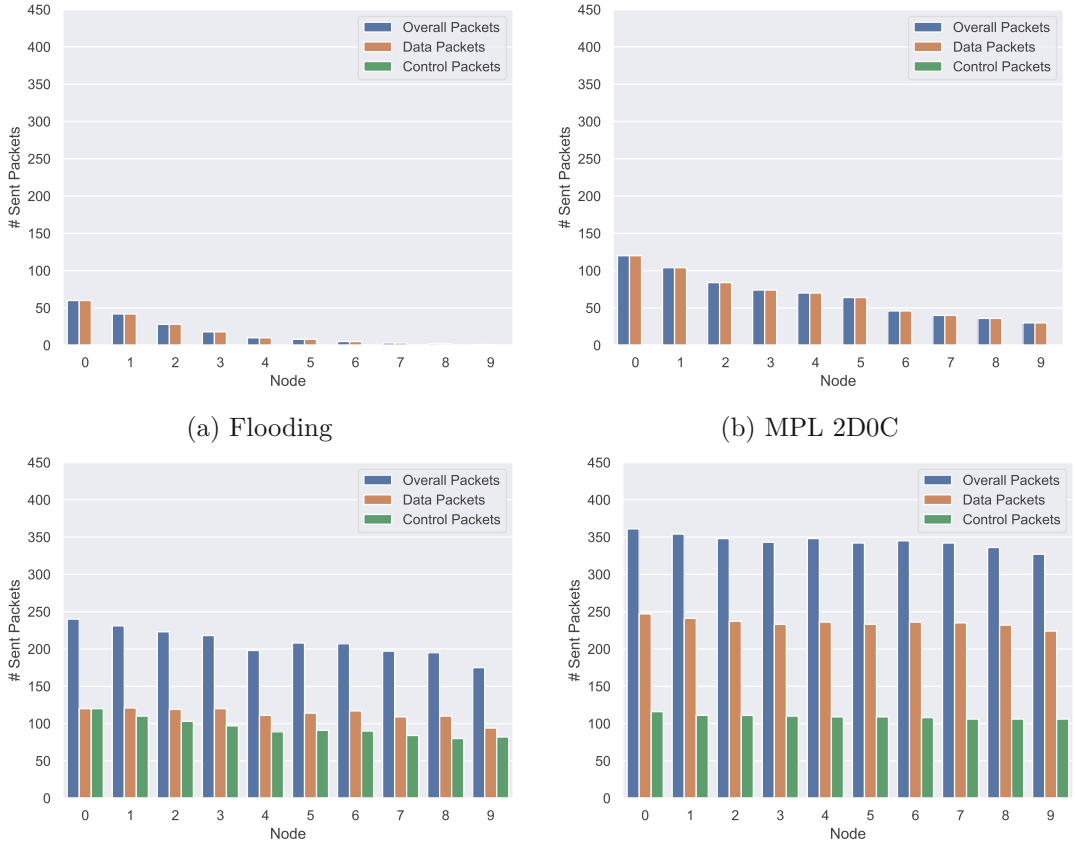
The performance of different protocol configurations are compared by calculating the metrics defined in section 5.4, which allow a comparison by using hardware-dependent and hardware-independent metrics.

6.2.1 Line Structure

The simulation results of the *Line* structure provide information on how well a routing protocol can transmit data if there is only a single path through the whole network. Thus, the routing protocol has to efficiently transmit all its messages to the nodes in this network. Especially for the simulations with the *Line* structure, we want to note that the simulations use a single-hop-PDR of about 75%. Therefore, the *Line* structure with a single receiving node is an extreme use-case for a network with unmovable nodes and limited PDR. This specific scenario and how different routing protocols handle it can be seen in Figure 6.1 for a *Line* structure of 10 nodes. It shows the number of sent messages for each node. The sender node is indicated by the leftmost bar and sends messages to the nodes on its right, indicated by the bars shown to the right. This sequence continues until the rightmost bar is reached. This number of sent messages is equivalent to the number of uniquely received data messages. Therefore, the sent messages of the ninth node indicate how many data messages reached this last node of the line topology and were successfully transmitted up to this point. This number of received messages then indicates how reliable the transmissions of a specific protocol configuration are.

The simulation results depicted in Figure 6.1 provide a good overview of how the different protocol settings affect the behaviour and actual performance of the whole network. In Figure 6.1a and Figure 6.1b, which show the number of messages each node sent using the *Flooding* and *MPL 2D0C* protocol, we can see that each hop decreases the number of messages that were successfully received and then forwarded to the next neighbour. For the plain *Flooding* protocol, this even means that the achieved PDR approximates to the configured 75%, so with each hop, we loose 25% of the messages a node forwards. This causes the last node to receive only a single data message out of the 60 that are initially sent. All the others were dropped during the previous hops.

Let's compare *Flooding* and *MPL 2D0C*. We observe that the *MPL 2D0C* routing protocol initially duplicates the 60 initial messages, such that the sender node initially transmits twice as many data messages as for the *Flooding* protocol. Therefore, achieving a PDR of approximately 75%, directly showing that if we send twice the amount of messages, the drop-rate is halved to 25%. Although, still only 7 of the 60 unique and initially sent messages are received by the last node, which shows that simply increasing the number of redundant messages increases the reliability, especially for the first few hops. However, the message delivery throughout the whole network is still very inhomogeneous. This problem can either be tackled by simply further increasing the amount of data messages the nodes send. This will further improve the number of delivered messages. However, the distribution will still be very unbalanced throughout the network. The second approach is to use the *reactive* mode of MPL, like shown in Figure 6.1c for the *MPL 2D1C* configuration. It only adds the *reactive* mode configured with a single control



(a) Flooding

(b) MPL 2D0C

(c) MPL 2D1C

(d) MPL 4D1C

Figure 6.1: Number sent packets for the *Line* structure with different configurations

message to the previously used *MPL 2D0C* configuration. As seen in Figure 6.1c, it provides a more even distribution of the sent and delivered messages throughout the network at the cost of sending nearly three times as many messages, data and control messages, throughout the whole network. The comparison between Figure 6.1b and Figure 6.1c shows the impact of the *reactive* mode on the network's behaviour as it tries to synchronize the data messages throughout the whole network by using its control messages to restart the *proactive* mode for certain messages. This resulted in better reliability with the obvious drawback of higher energy consumption due to the vastly increased number of sent and received messages.

Further, the *MPL 2D1C* protocol is still incapable of ensuring a completely reliable and even distribution of the data messages throughout the whole network due to unsuccessfully transmitted control and data messages. The only solution to achieve reliable transmissions from one to the other end of the whole line structure is to increase the amount of data packets our nodes send and accept the drawback of even higher energy consumption further. Our simulation results for the *MPL 2D1C* and *MPL2D2C* simulations shown in

Table 6.1 indicate that solely increasing the amount of sent control messages does not change the performance significantly. The results show that it is much more effective to keep the *reactive* mode enabled with a single control message and only increase the parameter for the *data-message expirations* to send more data messages. In Figure 6.1d the result for the *MPL 4D1C* simulation is depicted, which shows a very even distribution for the sent messages for both the data and control messages, which indicates that most of the initially sent packets were successfully transmitted to all the nodes inside the network. The high amount of successfully delivered messages can also be seen in Table 6.1, where we calculated a *message delivery ratio* of approximately 95.6% for the *MPL 4D1C* simulation. This means that out of the 540 (=9*60) messages that had to be delivered to the 9 other nodes that subscribed to the sender. Therefore, *MPL 4D1C* was able to successfully deliver 516 of these messages and only 24 were lost and did not reach their destinations, which is quite a good result for this very difficult and lossy *line* network topology.

Protocol	Sent Packets [#]	Received Packets [#]	Transmitted Bytes Overhead [%]
Flooding	174	287	0%
MPL 1D0C	159	257	0%
MPL 2D0C	706	1252	0%
MPL 1D1C	922	1613	11.8%
MPL 2D1C	1934	3472	146.5%
MPL 2D2C	2724	4891	217.6%
MPL 4D1C	3523	6352	389.0%

Protocol	Delivered Data Message Ratio [%]	Total Energy Consumption [J] (Energy Consumption without Baseline [J])	Energy per Delivered Data Message [J] (Energy without Baseline per Delivered Data Message [J])
Flooding	32.0%	626.612 (0.010)	19.6 (0.0003)
MPL 1D0C	29.3%	626 (0.113)	21.4 (0.0039)
MPL 2D0C	61.7%	634.088 (7.486)	10.3 (0.1214)
MPL 1D1C	54.4%	630.767 (4.165)	11.6 (0.0765)
MPL 2D1C	77.2%	640.526 (13.924)	8.3 (0.1803)
MPL 2D2C	85.6%	645.089 (18.487)	7.5 (0.2161)
MPL 4D1C	97.2%	654.795 (28.193)	6.7 (0.2900)

Table 6.1: Metrics to compare the protocols energy efficiency for the 10 node *Line* network

Table 6.1 shows the simulation results for the previously introduced metrics. These values indicate the energy consumption of the whole network and allow us to compare the different protocols against each other. The results obtained from these metrics confirm our interpretations drawn from the diagrams in Figure 6.1 the more packets our routing protocols are sending, the more of the 60 initially generated packets get delivered to all the nodes inside the network, which can be seen from the *Delivered Message Ratio* metric. This increase in the sent and received packets also increases the number of redundant data messages received by the nodes and the total overhead of the transmitted bytes, compared to a perfect transmission without any lost data.

Further, it is interesting to see that the *Transmitted Byte Overhead* metric is that the protocols that send the least amount of packets result in an overhead of 0% as they do

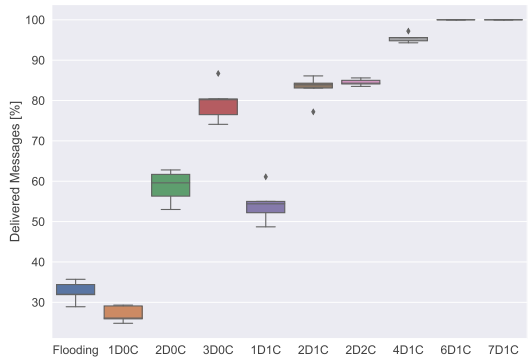
not even send enough bytes as it would be necessary to deliver the 60 initial packets to every single node. This shows, that the first three protocol configurations from Table 6.1 cannot even achieve a 100% *Delivered Message Ratio* for our setup of the *line* topology as they simply do not send enough messages, while having a too high drop-rate.

Another interesting aspect of these metrics is the impact and effect of the *reactive* mode. If we compare the results for the *MPL 1D0C*, *MPL 2D0C*, *MPL 1D1C* and *MPL 2D2C* we see, that the *reactive* mode affects the behaviour the most when it gets activated with a single sent control message. The effect of the *reactive* mode for the *MPL 1D1C* compared to *MPL 1D0C* improved the overall performance but also vastly increased the value for the *Redundant Received data messages* as its control messages re-trigger the data message transmissions of the *proactive* mode. This mode can distribute the packets further within the network, which led to a notable increase for the *Delivered Message Ratio* compared to the *MPL 1D0C* protocol. While the usage of a second control message expiration with the *MPL 2D2C* protocol causes only minimal changes for the network's overall *Delivered Data Message Ratio*, compared to the extra amount of energy it uses, as shown in Figure 6.2. Therefore, we can conclude that using only a single control message expiration provides the most improvement, and it is not beneficial to use more than that as it only increases the energy consumption but does not provide a significant improvement in the data message delivery ratio.

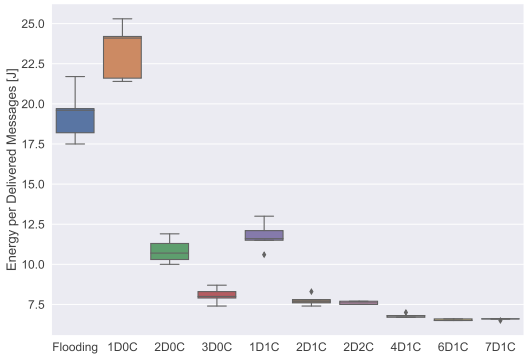
If we also consider the results from the *MPL 2D0C* simulation in this comparison, it can be seen that the effect of adding a second expiration for the *proactive* mode is nearly twice as effective as activating the *reactive* mode. Not only achieves *MPL 2D0C* an increase in the *Delivered Message Ratio* of $45.2\% - 23.9\% = +21.3\%$, which is approximately two times the improvement as the one obtained from *MPL 1D1C*, which gained only $34.6\% - 23.9\% = +10.7\%$. Additionally, the *MPL 2D0* routing protocol even used less energy than the *MPL 1D1C*, which again results in a far better result for the *Energy per Delivered Data Message* metric as depicted in Figure 6.2b.

Figure 6.2 compares several different protocol configurations for their energy efficiency and reliability by comparing their results for the *Delivered Data Message Ratio* and *Energy per Delivered Data Message* metric. As seen, the *Flooding*, *MPL 1D0*, *MPL 2D0C* and *MPL 1D1C* protocols are very inefficient, as they have a bad data message delivery ratio. This again leads causes a lot of used energy without getting any available reward from it. This relation between the consumed energy and delivered data messages are expressed with the *Energy per Delivered Data Message* metric, which indicates how much energy our network consumes on average to deliver a single data message to a node within the network.

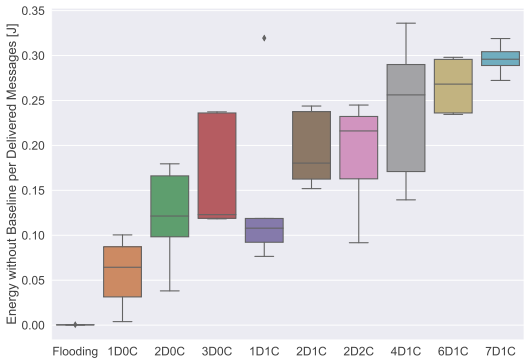
Another interesting conclusion from Figure 6.2b, is that only increasing the amount of sent messages is not the ideal way to increase performance and energy efficiency. It is more effective to enable the *reactive* mode with a single control message expiration than to keep it deactivated and only use additional data message expirations. This can be seen by comparing two pairs of simulations from Figure 6.2b, both *MPL 1D1C* and *MPL*



(a) Comparison of Delivered Messages Metric



(b) Energy per Delivered Messages with included Baseline Energy Consumption



(c) Energy per Delivered Messages with removed Baseline Energy Consumption

Figure 6.2: Energy Efficiency and Reliability of the Routing Protocols for the *Line* Topology

2D1C achieve a better performance for both metrics, than their direct counterparts *MPL 2D0C* and *MPL 3D0C*.

Further, the results from Figure 6.2b show that delivery performance is far more critical for our simulations than reducing the sent overhead. Especially for networks like the ones we have to use that do not use methods like RDC, IPv6 Time Slotted Channel Hopping (6TiSCH) [VWC+20] to reduce the energy consumption of the PHY during their long idle periods. This can be seen by comparing the results for the *MPL 6D1C* and *MPL 7D1C* protocols in Figure 6.2b. They seem identical, by both of them achieving a perfect 100% data packet delivery rate, with and an *Energy per Delivered Data Messages* of 6.5J and 6.6J.

Effectively the network using *MPL 7D1C* consumes 5.6J more than the one using *MPL 6D1C* due to the additional sending and receiving processes. Still, both achieve a nearly identical result for our energy efficiency metric. This behaviour originates from the incredible inefficient energy consumption of our LR-WPAN-PHY, for which we simulated

a total minimum energy consumption of approximately 62.6602J per 3800 seconds of simulation time. Accumulated for the whole network, this results in minimal energy consumption of 626.602J for our 10 nodes, without any sending activity. Compared to this enormous base energy consumption, the additionally consumed 5.6J are only 0.9% of additionally consumed energy. Suppose we extrapolate this percentage over a whole year. In that case, we lose approximately three days of additional runtime for an entire year, considering we can find battery sources that are durable enough to power our energy-hungry nodes for an entire year.

As this incredible inefficient baseline energy consumption offset is equally present for every node and every routing protocol configuration. Therefore, this constant offset only distorts the overall analysis of the energy efficiency, as every protocol that simply sends a huge amount of data packets and therefore uses even more energy, is always better rated than the energy-saving routing protocols. Therefore, this huge and misleading energy baseline of all the nodes is not considered anymore for the further work of this thesis, instead the energy-efficiency always considers the values with the this constant receiving baseline removed.

In Figure 6.2c, the *Energy per Delivered Message* metric was recalculated with this constant offset removed, which provides a completely different performance than previously depicted in Figure 6.2b, as now the energy efficiency really matters. As it can be seen with the baseline removed, the *Flooding*, *MPL 1D0C* and *MPL 1D1C* can be considered the most energy efficient ones, as they achieve the most successful transmissions for their consumed energy. Especially the *Flooding* protocol is surprisingly incredible efficient, as it never sends any redundant messages or additional control packets and every successfully transmitted packet immediately improves its efficiency vastly. All the other protocols simply send way to much overhead and cannot achieve the same efficiency than the *Flooding* approach. Although, if there is a minimum amount of delivered data messages required by the WSN's application, the *MPL 1D1C* and *MPL 2D1C* are a better choice as these protocols achieve relatively high ratios of delivered data messages, while at the same time also achieving a good and constant energy efficiency.

6.2.2 Circle Structure

The following network topology we simulate is the *circle* topology. The most important difference between a *Line* and *circle* structure is that every node within a *circle* has two neighbours it can send messages to. So generally, we can cut the *circle* open at the position of the initial sending node and handle it as a line structure, where we simultaneously send messages from both of its ends, which halves the theoretical number of hops necessary to reach the nodes in the middle of this transformed *line* structure. This vastly reduces the number of lost and dropped messages due to the fewer hops necessary to reach every single node. Additionally, a message that successfully reached the node in the middle is further forwarded into the second half of the line, so if the packet on the one side is lost earlier, there is still an additional possibility that the packet

from the other side reaches several nodes from the other half, which again increases the possibility that a packet is transmitted to as many nodes as possible.

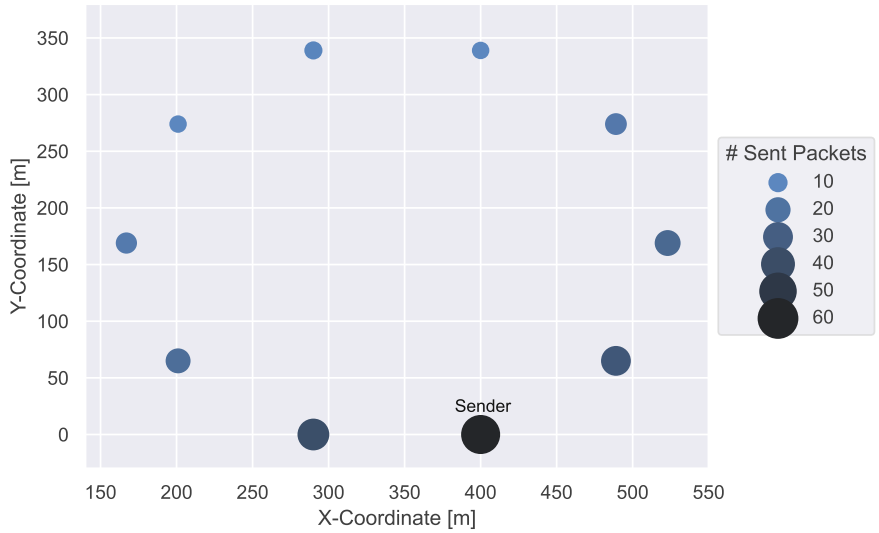


Figure 6.3: Sent data messages for *Flooding* in a 10 node circle structure

Figure 6.3 shows the results of the *Flooding* protocol used with a 10 node *circle* structure. Node 0 is the sender node, which simultaneously sends its packets to nodes 1 and 9. With every hop, the nodes get smaller and lighter as more and more messages get dropped, but still, every node receives at least ten different data messages, which is far better than the *Flooding* protocol achieved with the *Line* structure. This single simulation results impressively shows the influence a certain network topology has on a routing protocols performance. In case of the *Circle* structure, this improvement vastly comes from reduced amount of maximum hops till a packet reaches the nodes furthest away. Hence, by halving this maximum amount of hops and adding a redundant path to the node furthest away the *Flooding* protocol was able to achieve a *Delivered Data Message Ratio* of 58%, which is nearly twice as good as for the *Line* structure that only achieved 32%. The detailed results of the *Circle* topology are shown in Table 6.2.

Another notable aspect of the *Circle* structure's results compared to the results from the *Line* structure, is that the delivery rate for the MPL routing protocols for *Flooding* and *MPL 1D0C* doubles, while all the other ones do not profit that from the topology switch. These remaining protocols do not receive the same amount of performance increase than *Flooding* and *MPL 1D0C*, but still these protocols also increased their performance. *MPL 4D1C* even achieved to deliver every single message correctly, *MPL 2D1C* and *MPL 2D2C* both achieved a remarkable result of 95.7% delivered messages. Although, as for the *Line* topology, this good result with the *Circle* structure for the message delivery ratio comes with the drawback of a huge amount for the *Sent Bytes Overhead* metric. This additional overhead indicates, that *MPL 4D1C* sends approximately four times the minimum necessary amount of bytes as additional overhead.

Protocol	Sent Packets [#]	Received Packets [#]	Transmitted Bytes Overhead [%]
Flooding	260	516	0%
MPL 1D0C	258	516	0%
MPL 2D0C	984	1968	0%
MPL 1D1C	1210	2418	0%
MPL 2D1C	2286	4572	184.1%
MPL 2D2C	3051	6101	250.6%
MPL 4D1C	3565	7129	395.1%

Protocol	Delivered Data Message Ratio [%]	Total Energy Consumption [J] (Energy Consumption without Baseline [J])	Energy per Delivered Data Message [J] (Energy without Baseline per Delivered Data Message [J])
Flooding	57.8%	626.625 (0.023)	10.8 (0.0004)
MPL 1D0C	56.9%	630.055 (3.453)	11.1 (0.0607)
MPL 2D0C	88.1%	633.798 (7.195)	7.2 (0.0816)
MPL 1D1C	74.6%	631.695 (5.093)	8.5 (0.0682)
MPL 2D1C	96.9%	636.619 (10.017)	6.6 (0.1034)
MPL 2D2C	95.7%	643.386 (16.784)	6.7 (0.1753)
MPL 4D1C	100%	647.480 (20.878)	6.5 (0.2088)

Table 6.2: Metrics to compare the protocols energy efficiency for the 10 node *Circle* network

Overall the performance for the *Circle* structure is best described by the *Energy without Baseline per Delivered Message* metric, which indicates that *Flooding* is again by far the most energy efficient protocol, while *MPL2D2C* and *MPL 4D1C* can be considered completely energy inefficient as they generate way too much overhead without achieving better results than the *MPL2D1C* that achieves a similar *Delivered Data Message Ratio*, while at the same time achieving a much lower energy consumption.

6.2.3 Grid Structure

The following discussed network topology is the *Grid* structure, that consists out of 25 nodes that are organized into rows of 5. The results for the *Flooding* protocol which is again the most energy efficient one shown in Figure 6.4. Due to the network's layout and its increased density, each node on the network borders has at least two neighbours, while the ones on the inside all have exactly four neighbours. This increased network density leads to a much higher traffic for the whole network and causes an overall increase in the energy consumption of each node. As Figure 6.4 shows, the nodes on the right and upper edges receive the least amount of messages as they are the nodes furthest away from the sender and have the lowest number of neighbour, hence the possibilities to successfully receive a data packets are lower than for the other nodes.

The *Flooding* protocol, therefore, achieves an even better result than for the previous *Circle* structure and increases its *Delivered Data Message Ratio* up to 74.8% for a network with more than double the size as the one used with the *Circle* topology. In Table 6.3, the detailed results for the other simulations are shown. As for the previous simulation setups, the *Flooding* protocol is again the most energy-efficient one, as it generates the

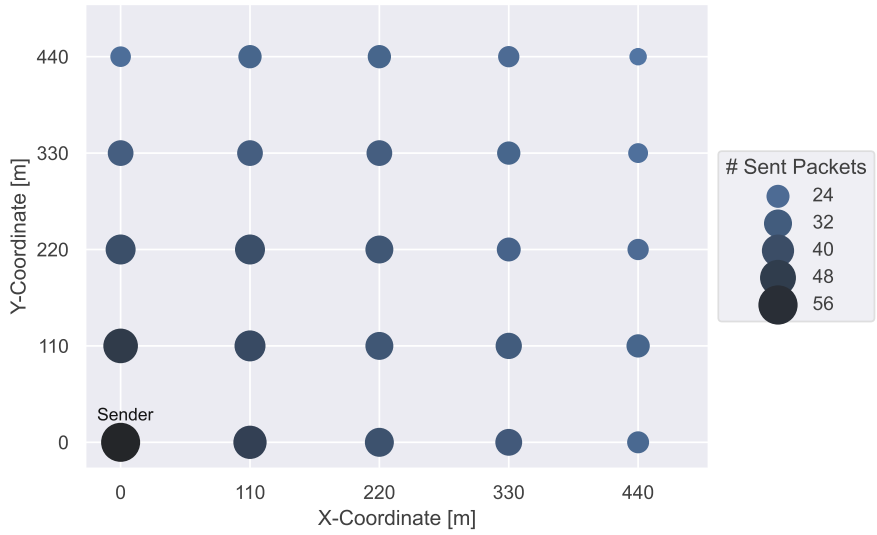


Figure 6.4: Sent data messages for *Flooding* in a 5×5 *Grid* structure

least amount of packets, which keeps the overall network traffic low and saves energy. Although, due to the high amount of redundant paths through the whole network, all the simulated routing protocols achieved a data message delivery ratio of at least 70%. Again as for the *Circle* topology, the *Flooding* protocol improved its performance the most and outperformed all the other routing protocol configurations again in terms of its energy efficiency.

Protocol	Sent Packets [#]	Received Packets [#]	Transmitted Bytes Overhead [%]
Flooding	890	2692	0%
MPL 1D0C	891	2872	0%
MPL 2D0C	2800	8963	0%
MPL 1D1C	3686	12049	0%
MPL 2D1C	5711	18225	184.9%
MPL 2D2C	7477	23761	243.2%
MPL 4D1C	8919	28507	396.0%

Protocol	Delivered Data Message Ratio [%]	Total Energy Consumption [J] (Energy Consumption without Baseline [J])	Energy per Delivered Data Message [J] (Energy without Baseline per Delivered Data Message [J])
Flooding	74.8%	1566.777 (0.271)	20.9 (0.0036)
MPL 1D0C	70.9%	1570 (3.647)	22.1 (0.0514)
MPL 2D0C	94.4%	1597.439 (30.934)	16.9 (0.3278)
MPL 1D1C	84.1%	1594.216 (27.711)	19.0 (0.3298)
MPL 2D1C	96.9%	1605.757 (39.252)	16.6 (0.4049)
MPL 2D2C	97.1%	1622.857 (56.352)	16.7 (0.5805)
MPL 4D1C	100%	1627.108 (60.603)	16.3 (0.6060)

Table 6.3: Metrics to compare the protocols energy efficiency for the 5×5 *Grid* network

The other simulation results are similar to the ones from the previous *Circle* topology.

These results again show that all the MPL routing protocol configurations need at least ten times more energy than the *Flooding* approach. Especially the MPL routing protocols that have the *reactive* mode enabled perform worse in terms of actual energy efficiency as they send too many messages that are not suppressed by the *polite gossiping* approach of MPL.

The overall conclusion from the simulations of the three basic network topologies *Line*, *Circle* and *Grid* is that *mesh-under Flooding* is by far the most energy-efficient protocol configuration from the ones evaluated. The only drawback of the *Flooding* protocol is its low message delivery ratio in very sparse networks like the *Line* structure. Although, it is reasonable to say that such a network topology is a difficult task for a low-power and lossy network that requires complex routing mechanisms like MPL to ensure a certain *Data Message Delivery Ratio*. These specific cases where a minimal amount of delivered data messages is required and *Flooding* cannot ensure this is the only reason to use a less energy-efficient routing protocol. In such a specific case with a minimum transmission performance required, the only routing protocols feasible to use are the ones that do not create a vast amount of overhead and send only the bare minimum needed to achieve the requirements. Such protocol configurations are the *MPL 2D0C* and *MPL 1D1C*. Using *MPL 1D0C* does not provide additional advantages as it achieves the same packet delivery performance as *Flooding* at worse energy efficiency.

6.2.4 Effect of the Redundancy Constant K

Selecting a suitable value for the *Redundancy Constant K* is a substantial part of the configuration of the MPL routing protocol. It defines how often a node has to receive the same data- or a consistent control-message until its scheduling of this data or control message is suspended, its message is not sent, and its expirations-counter is increased.

Therefore, this *Redundancy Constant K* directly controls the behaviour of the *polite gossiping* of MPL. When configured at a too high value, it affects the protocols *energy efficiency* as the already known messages are always transmitted and not suspended if they are re-received. If this parameter is too low, messages that have to be forwarded might get suppressed in certain situations.

As stated by the specification of MPL [HK16], the default value of K is set to 1 for both operation modes. The parameter K is often only relevant for specific topologies or transient situations, where it might be beneficial to use a higher value. Especially with networks that consist of randomly distributed nodes, due to this random distribution, not all the nodes may have an equal number of neighbours. Such a setup may lead to unwanted suppressions of packets that shall be forwarded.

Therefore, several simulations with the *Grid* structure were performed to evaluate the effect of changing the redundancy constant for both the *proactive* and *reactive* operation mode. The simulations used were based on the *MPL 2D0C*, *MPL 2D2C*, and *MPL 4D1C* variated K from 1 to 4 for these routing protocol configurations. Nonetheless, none of these additional simulation results showed any improvements or changes to the previously

described results in Table 6.3, which justifies the the possibly higher energy consumption and more network traffic. Hence, we derive from these results that it is ok to commonly set K to 1 and expect the network to work efficiently.

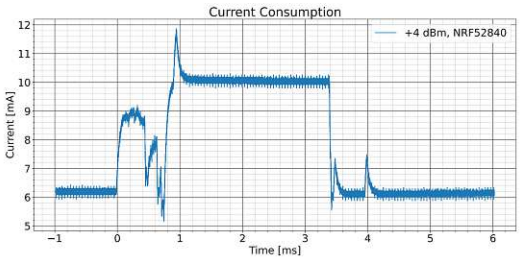
6.2.5 Energy Simulation vs. Real-World Measurement

The simulations and the results acquired from them allow for fast development and testing of complex hardware setups. However, these results are virtual simulations, hence approximations and abstractions of real-world hardware. These approximations raise the essential question of how accurate the provided results of the simulations are.

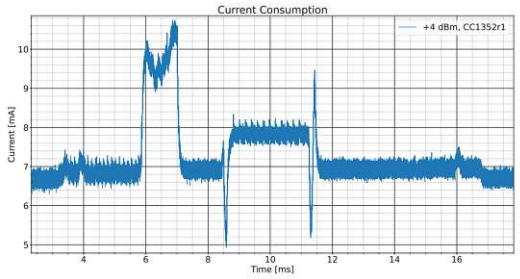
The results and measurements published by Adelmann [Ade21] are used to compare the simulation results to measurements obtained from physical hardware setups. This comparison between the simulator domain and actual hardware concentrates on the simulator parts necessary to determine the energy consumption of a simulated node. The first step, therefore, is to compare the simulator's utilized sending-current model to the current consumption of physical hardware nodes. In Figure 6.5, the current consumption during the sending process of the *NRF52840* and *CC1352r1* hardware nodes [Ade21], and the abstracted sending current model used by NS-3 to model and calculate the energy consumption of a virtual node.

As seen in Figure 6.5c, the current model used by NS-3 is a completely simplified rough abstraction of the real-world current-consumption models like Figure 6.5b and Figure 6.5a. The simulator's energy model only consists of constant current values applied to the durations of the LR-WPAN PHY's states. Comparing NS-3's current model with the measurements of the two hardware nodes shows, that there are several crucial differences between them which show that the simulator does not achieve a proximate result for the energy consumption of a real-world hardware node.

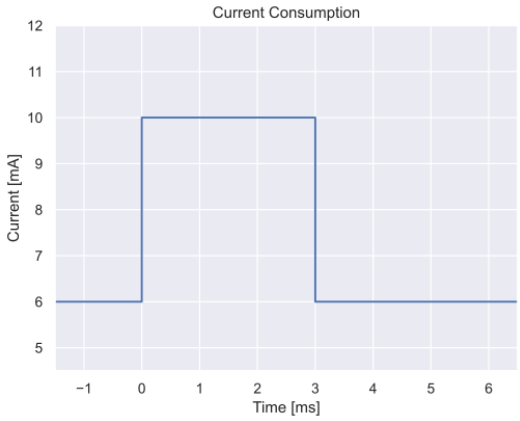
This comparison shows that the need for better simulation models is inevitable to improve the resulting quality of the energy simulations of NS-3. It seems nearly impossible to generate a completely generic current model, applicable and configurable to match the behaviour of various hardware nodes. Additionally, real hardware has even more physical properties, like chip ageing, temperature effects and many more, that shall be considered when aiming to develop an ideal simulation model. Another major part that adds up to a node's energy consumption is the necessary amount of energy consumed by the Central Processing Unit (CPU). As shown in the work of Adelmann [Ade21], the energy consumption of the CPU is a substantial part of the overall energy consumption and is again very different across the various hardware platforms. For some nodes like the *CC1352*, his work showed that the actual software executed by the CPU is responsible for one-third of the energy consumption during a sending process. The energy consumption by the CPU shows that even the program executed on the CPU has a remarkable influence on the energy consumption of a node. This finding introduces an additional factor that affects a node's energy consumption. Hence, also the computational complexity of the used routing protocols like AODV6 or MPL, which have a much higher complexity and



(a) Send Current of *NRF52840* node [Ade21]



(b) Send Current of *CC1352r1* node [Ade21]



(c) Send Current Model of the NS-3 simulator

Figure 6.5: Sending currents of two hardware nodes and the sending energy model used by NS-3

memory consumption, than a simple *Flooding* implementation. Therefore, it is reasonable to expect the *Flooding* protocol to be even more energy efficient than indicated by the simulations described in section 6.2.

This finding on the importance of the energy consumed by the processor indicates even more crucial to keep the number of sent messages as low as possible to minimize the amount of time the CPU, *sender* and *receiver* units of the node have to leave their idle states and switch on. Apart from the complex current consumption of the network PHY, an exact energy-simulator simulator has to simulate a node's processor and even the executed program. Accordingly, a realistic energy simulation may only be possible to acquire when using an exact hardware emulator, which can be fine-trimmed to all the specifics of a specific node. In comparison, a generic and abstract network simulator like NS-3, whose aim is to simulate a broad range of communication networks and their numerous protocols. Hence, it is not NS-3's primary use case to perform high-end energy simulations. Instead, it uses a very abstract and simplified current model, which is easy to configure and use and aims to provide a rough and quick estimation of the performance and energy consumption of a specific network setup. It is not the intended use case of NS-3 to provide high-end energy simulations for specific hardware nodes that are

directly mappable to the real world. Of course, there is always potential and a need for better simulation models. Such an example is the need for an improved LR-WPAN model. Especially for the PHY- and MAC-layers, which need crucial improvements to implement more detailed energy models for LR-WPAN, that allows simulating more complex PHY behaviour like realistic switching effects, when enabling or shutting down the sender or receiver. Such an improvement would create the necessary base-features to allow an implementation of an RDC functionality, which improves the overall quality and applicability of the simulation results for the energy consumption.

6.3 Network Lifetime Simulations

The next category of performed simulations is the ones that utilize the shut-off functionality of NS-3's LR-WPAN energy model. Other than the previously discussed simulations from section 6.2, where the nodes had an infinite amount of energy available, the simulations performed in this section only provide a limited amount of energy to the nodes, which is not enough for all the nodes to last for the whole simulation. Therefore, nodes that deplete their energy source switch off their sender and receiver units and do not interact with the remaining active nodes. These simulations allow us to detect which parts of a network use the most energy and deplete their energy fastest. The information from these simulations is also helpful for detecting nodes that need to use batteries with a higher energy capacity or even utilize mechanisms like energy harvesting to reload their strained batteries.

These simulations use the *Network Lifetime* and *Normalized Network Lifetime per Delivered Data Message* metrics to quantify and compare the behaviour for the different routing protocols and network topologies. The used *Network Lifetime* metric is a three-value tuple sampled in three different situations. The first-lifetime value is sampled when the first node within the network is depleted. The second and third ones are taken after 50%, and 90% of the nodes have shut down. These three values allow a better insight into the temporal behaviour of the network's depletion process.

Table 6.4 shows the results for the different routing protocol configurations obtained from the network simulations using nodes with a limited energy source. The first thing to notice from these results is the minimal time difference between the three values of the metric's tuple. This minimal time difference indicates that the overall network becomes dysfunctional within a timespan of only a few milliseconds, regardless of the utilized routing protocol. This behaviour again origins in the functionality of the LR-WPAN network PHY used to simulate the nodes, as NS-3's PHY implementation is not capable disable the receiver to reduce the energy consumption. As shown previously in section 6.2, this permanently enabled receiver drains the vast amount of energy from the battery compared to the part used to transmit messages to other nodes. Hence, all the nodes within the network deplete nearly simultaneously as the nodes' differences in the amount of energy used to send their messages are only a minimal fraction compared to the energy consumed by the permanently enabled receiver.

Topology	Line			Circle			Square		
Network-Lifetime	1^{st} [s]	50% [s]	90% [s]	1^{st} [s]	50% [s]	90% [s]	1^{st} [s]	50% [s]	90% [s]
Flooding	3389.66	3389.67	3389.68	3389.61	3389.62	3389.62	3389.24	3389.25	3389.26
MPL 1D0C	3389.67	3389.69	3389.69	3389.58	3389.60	3389.60	3389.03	3389.04	3389.04
MPL 2D0C	3389.17	3389.19	3389.20	3388.97	3388.98	3388.98	3387.29	3387.29	3387.29
MPL 1D1C	3389.15	3389.17	3389.18	3388.98	3388.99	3388.99	3387.53	3387.53	3387.54
MPL 2D1C	3388.35	3388.36	3388.36	3388.24	3388.24	3388.24	3385.88	3385.88	3385.89
MPL 2D2C	3388.05	3388.07	3388.07	3387.85	3387.85	3387.85	3384.99	3385.00	3385.00

Table 6.4: Results for the *Network-Lifetime* simulations of the three different network topologies when the 1^{st}, 50% and 90% of a networks' nodes shut down

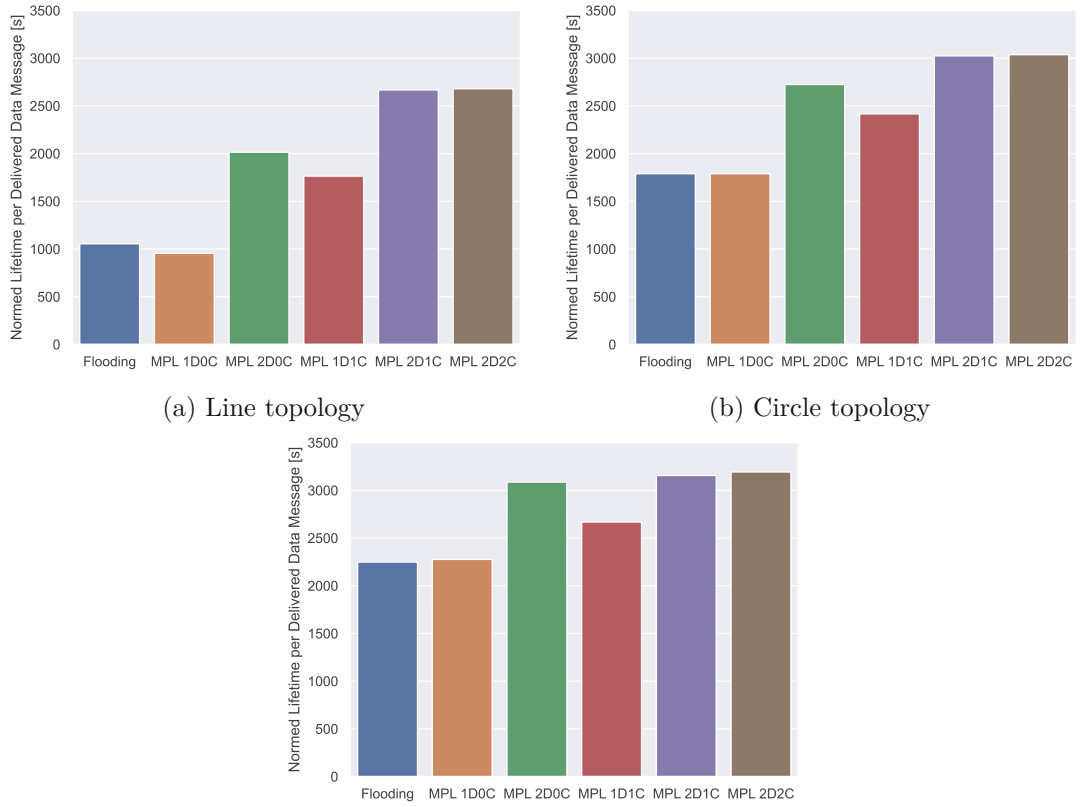
Nonetheless, the results still allow an interpretation of the energy consumption of a routing protocol used within the *Line*, *Circle* and *Grid* topology. Table 6.4 combined with the results from section 6.2, shows that the more messages are transmitted within a network, the lower the overall network lifetime gets. This effect is the biggest for the *Grid* topology as a node sends a higher number of messages than with the other topologies. This higher number of transmitted messages a node sends to its multiple neighbours results in more successfully sent messages. Hence, when comparing the *Network Lifetime* of the *Flooding* protocol between the *Line* and *Grid* topologies, the nodes within the grid-formed network send and receive more messages than the ones from the *Line* topology, which results in a lower *Network Lifetime*. The same logic can be applied when comparing the *Network Lifetime* results of the different routing protocols for the same network topology.

When comparing the results for the *Grid* topology simulations, it is noteworthy that, apart from the permanently enabled receiver, the used routing protocols still make a difference for the *Network Lifetime*. For the simulations using the *Grid* topology, this makes a difference of up to 4.26 seconds between the *Flooding* and *MPL 2D2C* routing protocol during an approximate runtime of one hour. When scaling this result up to a maintenance cycle of a whole year until the nodes batteries have to be changed, these 4.26 seconds scale upwards to approximately 10 hours of lifetime difference between the *Flooding* and *MPL 2D2C* routing protocol. This estimated time difference does not manifest in a substantial difference in the application lifetime but inferred that these results represent the worst case where the receiver or sender is always enabled. Hence, as soon as there is some functional RDC mechanism used that balances the ratio between energy used to send and energy consumed to power the receiver, the importance of the used routing protocol on the *Network Lifetime* will vastly increase and resulting in higher differences in the *Netowork Lifetime*.

Although the *Network Lifetime* itself gives us a good indication of how long a certain amount of nodes stays operational within a network, it does not provide us with any information if these active nodes are actually receiving any new messages or if they are only idling until their battery depletes too. To better understand how performant such a long-living network is, the *Network Lifetime* metric is combined with the amount of at

least once received messages. This newly created metric combines the *Network Lifetime* with a performance metric that allows distinguishing networks that successfully send and receive data or only idle without any activity.

In Figure 6.6, the simulation results for the *Network Lifetime per Delivered Data Message* normalized to the overall network size are shown. These results are used to quantify the performance of the different routing protocols for their energy efficiency, paired with the amount of successfully delivered messages. In Figure 6.6a, the results for the *Line* topology are shown. The results shown are contrary to the ones listed in Table 6.4, where the *Flooding* protocol achieved the highest *Network Lifetime*. Although, when combining the *Network Lifetime* energy-efficiency metric with a transmission metric, this indicates the effective *Network Lifetime*. The whole situation changes, as the results are shown in Figure 6.6a and Figure 6.6b for the *Flooding* and *MPL 1D0C* routing protocols indicate that they were only able to achieve such a high *Network Lifetime* because the nodes were idle most of the time.



(a) Line topology

(b) Circle topology

(c) Grid topology

Figure 6.6: Results for the *Normalized Network Lifetime per Delivered Data Message*

Therefore, this metric can quantify how energy efficient and performant a routing protocol configuration is for a specific network topology. For the results obtained from

the performed NS-3 simulations, this metric indicates that the nodes sending and receiving the most messages are also the ones performing best. This result is distorted again by the fact that the amount of energy used to send messages is irrelevant compared to the energy consumed by the PHY's receiver. Due to this limitation, the routing protocols have only a minimal difference in their achieved *Network Lifetime*, which causes the amount of successfully delivered data messages to the relevant aspect that decides which routing protocol works best. Nonetheless, this metric still provides the correct information on which routing protocol configuration is most energy-efficient and achieves the best compromise between delivery performance and energy consumption.

6.4 Interference Simulations

The final set of simulations probes the different routing protocols for their ability to overcome high-interference situations. In these situations, the physical transmission medium is highly distorted, which increases the possibility of irregular transmission failures. Interference from other nodes' message transmission is a common problem with wireless networks that depending on the amount of interference created, can vastly influence the PDR in a WSN. Therefore, the performed *interference simulations* aim to evaluate the effect of interference distortions from other WSN networks on the performance of a routing protocol configuration for a specific network topology. The *interference simulations* utilize similar topologies and setups as the previously described simulations from section 6.2. The only differences to these topologies are the so-called *interference nodes* and an increased sending power of +4 dBm, such that all of the observed transmission failures occur due to interference. The additional interference nodes form a separate LR-WPAN network that communicates on the same channel and periodically sends packages between them to create traffic on the physical medium that distorts the transmissions from the *main*-network. The conducted simulations use exactly two *interference nodes*, one sender and one receiver. The interference nodes are configured according to the parameters listed in Table 5.1.

The main goal of these simulations is to show how interference-tolerant the different routing protocol configurations are for the three main topologies. These interference simulations do not provide us with relevant energy consumption results, as the interference nodes send on the same frequency channel as the main network and therefore keep the receivers nearly permanently receiving the sent interference messages, which falsifies the values for the energy consumption. Hence, the metrics used in these simulations are the number of transmitted packets and the delivery ratio of the senders' different messages. These two metrics indicate best how much impact the interference packets make on the otherwise successful LR-WPAN transmissions.

The first simulation results are shown in Figure 6.7, which depicts the *Line* topology network with its ten nodes and the two additional interference nodes and their transmission range, indicated by the two red circles. These circles are indications to mark which nodes have to deal with the most interference distortions.

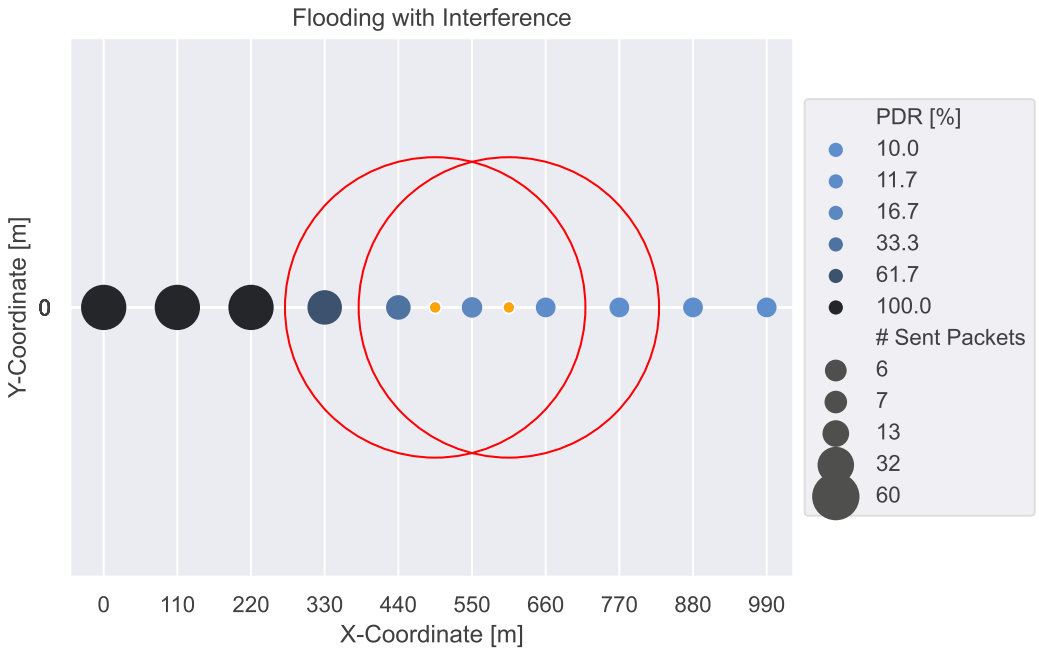


Figure 6.7: Impact of interference on the *Flooding* protocol for the *Line* topology

The graphic shows that, the first hops at the beginning are both done without any lost packet, as both following nodes receive every one of the initially sent data messages. Although, as soon as the transmissions reach the first node, positioned within the transmission range of the interference-generating nodes, the amount of delivered messages suddenly declines. This drop in the PDR and the amount of sent messages indicates that the interference nodes have a devastating effect on the transmission quality of the remaining nodes. In the depicted situation of a *Line* topology with initially perfect transmissions, this impact leads to nearly 90% of the data messages dropped that do not reach the last node within the network. This result indicates that although LR-WPAN uses an CSMA/CA approach that checks if the medium is already used by another node, there are still lots of collisions and dropped packets. Due to the *Flooding* protocol not retransmitting unsuccessfully transmitted messages, the PDR drops to only 10%.

As a counterpart to the bad performance of the *Flooding* protocol, Figure 6.8 depicts the results for the *Line* topology used with the *MPL 2D1C* routing protocol configuration. These results for the *MPL 2D1C* show much better performance than the one achieved by *Flooding*. *MPL 2D1C* achieves a more than four times better PDR of the initially sent data messages. This result shows that *MPL 2D1C* utilizing the *reactive* operation mode and sending more than two times the amount of initially sent data messages is still unable to deliver at least 50% of the necessary data messages. Therefore, the only way to further improve the PDR in these interference setups for the *Line* topology is to increase the number of retransmissions of the data messages, which improves the odds

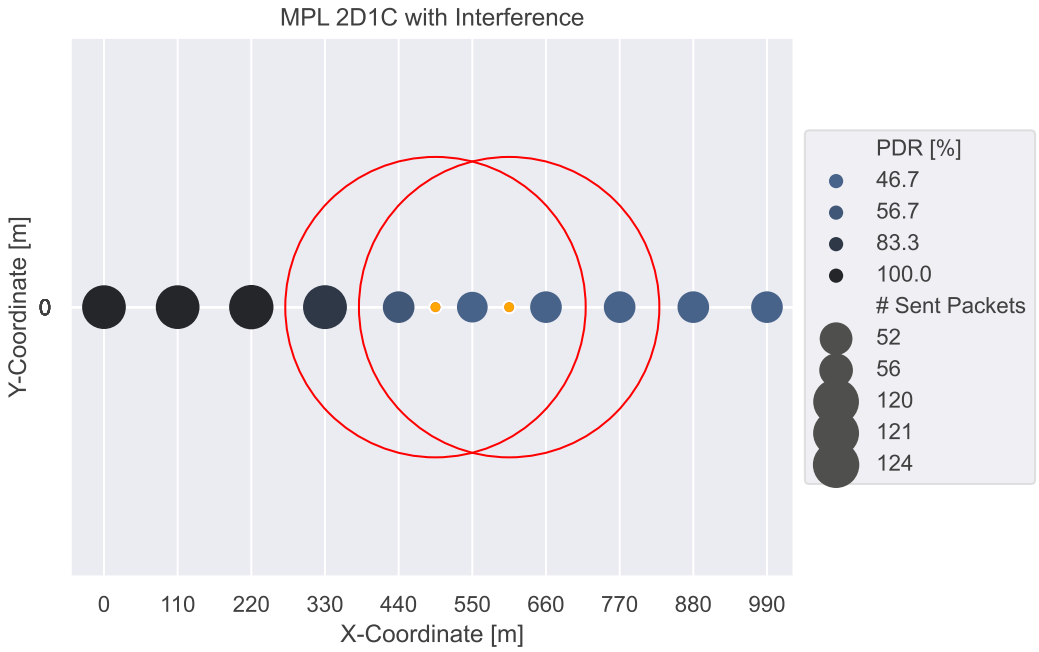


Figure 6.8: Impact of interference on the *MPL 2D1C* protocol for the *Line* topology

that one of a message's retransmissions is successfully delivered. On the other hand, such an increase automatically leads to additional energy consumption by the nodes and even more traffic and interference on the transmission medium.

At the moment, the described simulations only considered the *Line* topology consisting of one single path to transmit a message to any node within the network. If this path then becomes distorted because of any possible reason, the PDR of the whole network is reduced. Hence, it is also necessary to evaluate how a denser network like the *Grid* topology handles such an interference situation. Figure 6.9 shows such a case, where the interference-generating nodes are positioned right in the middle of the network. Although all the nodes are configured the same as they were for the previous simulations with the *Line* topology, the *MPL 1D0C* routing protocol can deliver more than 95% to any node within the network. This result again shows that the overall layout and topology of the WSN can be way more important than selecting the utilized routing protocol. Figure 6.9 even shows that although the nodes in the centre do receive nearly all of the different messages, they do not re-transmit them again due to the *polite gossiping* mechanism of MPL.

Hence, these conducted interference simulations showed that the overall layout of a WSN is the most important property that controls its behaviour. To make a WSN resilient to heavy distortions on the wireless medium or any other faults, mandatory to add multiple redundant paths within the overall network. Using such a dense network instead of a sparse one like the *Line* topology even allows using a simple and energy-efficient

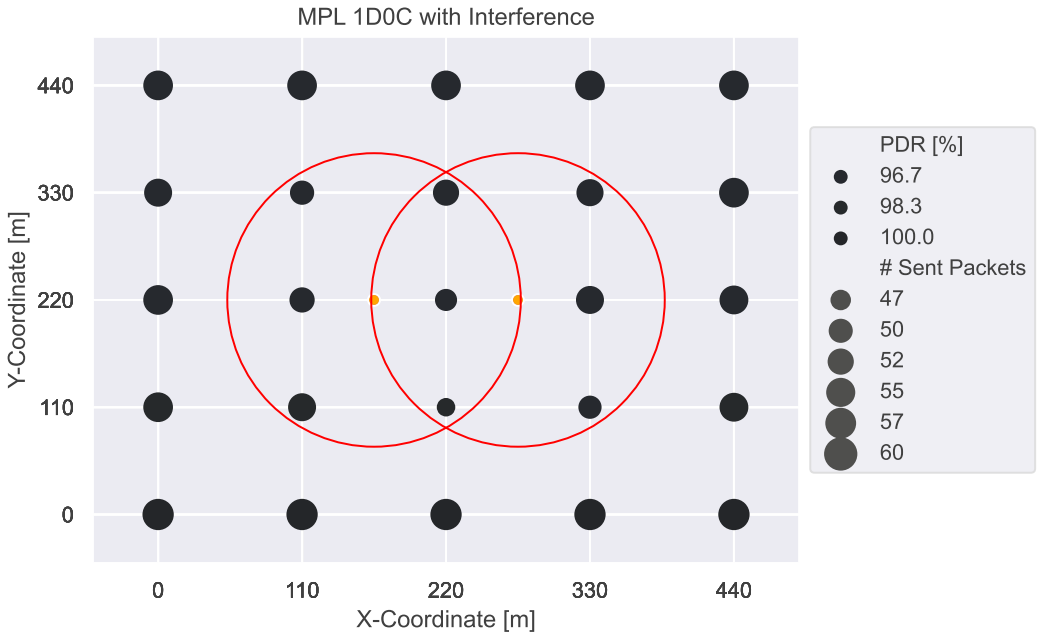


Figure 6.9: Interference simulation of the *MPL 1D0C* protocol for the *Grid* topology

routing protocol like *Flooding* or *MPL 1D0C* without any critical impacts on the overall transmission performance.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

This thesis analyses, implements, simulates and evaluates different multicast routing protocols in the simulation domain. Several network simulators are evaluated for their available simulation models and features to simulate crucial aspects of WSNs and finally the NS-3 simulator is selected for the further usage. Missing features like the MPL routing protocol are implemented. Several *hardware-independent* and *hardware-dependent* metrics are used to compare the simulation results of the different routing protocol configurations for their energy efficiency.

When applying these metrics on the obtained simulation results. It got obvious that due to the simplifications applied within the MAC layer of the currently available LR-WPAN simulation model of NS-3, it is not possible to use more sophisticated *Energy* simulation models. Due to these limitations of the used simulation models, it is not possible to directly map the results obtained for the *hardware-dependent* metrics from the virtually performed simulations to measurements taken from real-world hardware nodes. There is simply no such energy consumption model at the moment, that is generally applicable and versatile enough to describe the hardware nodes with all the necessary details to directly apply the *hardware-dependent* results form the simulations to real-world nodes.

Nonetheless, there are also the *hardware-independent* metrics, which showed, that their results can be used for reasonable comparisons between the virtual simulations networks and the real-world WSNs. Although, as shown by Adelmann [Ade21], the applicability of *hardware-independent* metrics is not available with every type of hardware. Hence, it solely depends on the actually used hardware setup for the nodes, if the *hardware-independent* metrics obtained per simulation can be used to describe the energy consumption of this node or not.

These conclusions and results of this thesis show, that simulations are a great tool to test and experiment with a WSN. Simulations allow a user to determine how certain network parameters, like different network topologies and interference, affects the overall behaviour of the network. Nonetheless, one has to be aware that a simulation is always a simplified abstraction of the real-world, that requires trade-offs between the achieved accuracy and feasibility of the simulation model. Hence, this thesis showed, that NS-3 has a major need for better *LR-WPAN* and *Energy* models in order to realistically simulate the energy consumption of a hardware node.

7.2 Future Work

As it is sheer impossible to address every detail, possibility and aspect that combines the domain of WSNs and the vital role they have in the global perspective of an IoT. Nonetheless, this thesis provides an entry point for researchers interested in *multicast routing protocols* used together with *Low Power and Lossy Networks*. Therefore, it is necessary to point out further aspects that could not be handled in this thesis, as the amount of work would have vastly exceeded the scope of this master thesis. Such aspects for *Future Work* are, for example, the additional implementations of other multicast routing protocols. Most importantly, if NS-3 finally releases an RPL module, which would level the ground for the implementation of further commonly used multicast routing protocols like SMRF and further enhancements of this routing protocol [OP12]. Further work may also include additional metrics, simulations with movable nodes and definitions for energy efficiency. Also, we can extend the existing simulation models to feature multiple message generator nodes within a single network or the behaviour and influence of interference generated by two concurrent networks instead of using artificial interference generating nodes.

Hence, it is still a lot of future work to do, especially with the tremendous current development process and speed of microelectronic devices and their used software. These new possibilities re-open the research domain of WSNs and combine their encapsulated view with a global perspective of the dynamic and highly connected IoT, which changes the handling and characteristic of specific properties like the exact causes and results for a node's energy consumption. All these significant changes in these previously well-known properties and the general consequences of integrating WSNs directly into the highly connected and dynamic world of the IoT provide this research domain with many further fascinating questions and problems to analyse and solve.

List of Figures

List of Tables

Glossary

6LoWPAN IPv6 over Low power Wireless Personal Area Network. xiii, 4, 10, 13–16, 20–22, 25, 35, 37, 38, 45, 91

CSMA/CA Carrier Sense Multiple Access-Collision Avoidance. 25, 85

GUI Graphical User Interface. 10–14, 16, 91

ICMPv6 Internet Control Message Protocol for the Internet Protocol Version 6. 29, 30, 33

IPv4 Internet Protocol Version 4. xiii, 35

IPv6 Internet Protocol Version 6. xiii, 1, 10, 13–15, 25, 26, 28–30, 35, 37–40

NS-2 Network Simulator Version 2. 6, 14–17

NS-3 Network Simulator Version 3. xiii, 6, 7, 9, 14–23, 37, 38, 40, 44–46, 48–53, 58, 65, 67, 79–81, 84, 89–91

OMNeT++ Objective Modular Network Testbed in C++. 6, 9, 12–17

OTcl Object Oriented Tcl. 14

RFC Request For Comments. 29

Tcl Tool Command Language. 14

TCP Transport Control Protocol. 33, 34

TTL Time To Live. 24, 35, 63

UDP User Datagram Protocol. 34

Acronyms

6TiSCH IPv6 Time Slotted Channel Hopping. 73

AODV Ad hoc On-Demand Distance Vector. xiii, 5, 7, 23, 35–38, 67, 68

AODV6 Ad hoc On-Demand Distance Vector (AODV) Routing for IP version 6. xiii, 7, 23, 37, 38, 49, 67, 68, 79

API Application Interface. 17

ARP Address Resolution Protocol. 38

BFS Breath First Search. 35

CPU Central Processing Unit. 79, 80

CSV Comma Separated Values. 49, 52

DEVS Discrete Event Simulator. 10, 11

HBA Home and Building Automation. 6

ICMP Internet Control Message Protocol. 44

IoT Internet of Things. xiii, 1, 4, 10, 15, 35, 90

IP Internet Protocol. 30

LEACH Low-Energy Adaptive Clustering Hierarchy. 5, 23

LOADng Lightweight On-demand Ad hoc Distance-vector Routing Protocol—Next Generation. 23

LR-WPAN Low-Rate Wireless Personal Area Networks. 7, 10, 12, 15, 20, 35, 37, 38, 45–48, 50, 51, 56–58, 65, 73, 79, 81, 84, 85, 89–91, 93

MAC Medium Access Control. 25, 65, 81, 89

MANET Mobile Ad-hoc Network. 5, 6, 35, 67, 68

MPL Multicast Protocol for Low-Power and Lossy Networks. xiii, 4, 5, 7, 10, 20, 23, 26–33, 35, 37–41, 43, 52, 62, 63, 67–69, 75, 78, 79, 86, 89, 91, 93

MSPSim MSP-Simulator. 10

NDISC Neighbour Discovery Cache for IPv6. 38

OS Operating System. 10

PDR Packet Delivery Ratio. 57–60, 62, 68, 69, 84–86, 91

PER Packet Error Ratio. 59

PHY Physical Layer. 6, 45–47, 50, 51, 65, 73, 79–81, 84

QoS Quality of Service. 5

RDC Radio Duty Cycling. 47, 65, 73, 81, 82

RERR Route Error. 36, 38, 68

RNG Random Number Generator. 19

RPL Routing Protocol for Low power and Lossy Networks. 5, 10, 15, 23, 90

RREP Route Reply. 35, 36, 38

RREP-ACK Route Reply Acknowledgment. 38

RREQ Route Request. 35, 36, 38, 68

SMRF Stateless Multicast RPL Forwarding. 23, 90

UML Unified Modeling Language. 45, 91

WSN Wireless Sensor Network. xiii, 1–7, 9, 10, 12–15, 20, 23, 24, 31, 33, 35–38, 46, 49, 55, 56, 58–62, 64, 65, 67, 68, 74, 84, 86, 89, 90

Bibliography

[Ade21] Stefan Adelmann. *A Survey of Energy Efficient Multicast Routing Protocols for Wireless Low Power and Constrained Devices*. Masters's thesis, TU Wien, 2021. https://repositum.tuwien.at/handle/20.500.12708/19215.

[AK19] Hayder Al-Kashoash. *Congestion Control for 6LoWPAN Wireless Sensor Networks: Toward the Internet of Things*. Springer theses. Springer, 2019.

[BMC+19] Michel Bakni, Luis Manuel, Moreno Chacón, Yudith Cardinale, Guillaume Terrasson, and Octavian Curea. WSN Simulators Evaluation: an Approach Focusing on Energy Awareness. *International Journal of Wireless & Mobile Networks*, 11(6):1–20, December 2019.

[BMLG17] Jimmy Bondu, Anupal Mishra, Vijay Laxmi, and Manoj Singh Gaur. Flooding in Secure Wireless Sensor Networks: Student Contribution. In *Proceedings of the 10th International Conference on Security of Information and Networks*, pages 151–156, Jaipur India, October 2017. ACM.

[DC98] Dr. Steve E. Deering and Alex Conta. *Generic Packet Tunneling in IPv6 Specification*. RFC2473. Internet Engineering Task Force, 1998.

[DH98] Stephen E. Deering and Robert M. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. Internet Engineering Task Force, 1998.

[EOF+09] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. Cooja/mspsim: Interoperability testing for wireless sensor networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, Brussels, BEL, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[GMG07] Antonio Garcia-Macias and Javier Gomez. MANET versus WSN. In Nitaigour P. Mahalik, editor, *Sensor Networks and Configuration: Fundamentals, Standards, Platforms, and Applications*, pages 369–388. Springer, Berlin, Heidelberg, 2007.

[Hab02] Brian Haberman. *Allocation Guidelines for IPv6 Multicast Addresses*. RFC 3307. Internet Engineering Task Force, 2002.

[HK16] Jonathan W. Hui and Richard Kelsey. *Multicast Protocol for Low-Power and Lossy Networks (MPL)*. RFC 7731. Internet Engineering Task Force, 2016.

[HSPDDCGL+20] Ángela Hernández-Solana, David Pérez-Díaz-De-Cerio, Mario García-Lozano, Antonio Valdovinos Bardají, and José-Luis Valenzuela. Bluetooth Mesh Analysis, Issues, and Challenges. *IEEE Access*, 8:53784–53800, 2020.

[IEE20] IEEE. IEEE Standard for Low-Rate Wireless Networks. *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, pages 1–800, July 2020.

[IH12] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. Springer US, Boston, MA, 2012.

[Kas20] Mohamed Rawidean Mohd Kassim. IoT Applications in Smart Agriculture: Issues and Challenges. In *2020 IEEE Conference on Open Systems (ICOS)*, pages 19–24, Kota Kinabalu, Malaysia, November 2020. IEEE.

[KB01] Luke Klein-Berndt. A Quick Guide to AODV Routing. `http://cs.uccs.edu/~chow/pub/master/pjfong/UCCS%20Project/Papers/A%2520Quick%2520Guide%2520to%2520AODV%2520Routing.pdf` Wireless Communications Technologies Group, NIST, 2001. Accessed: 2022-02-26.

[KHN14] Muhammad Amir Khan, Halabi Hasbullah, and Babar Nazir. Recent Open Source Wireless Sensor Network Supporting Simulators: A Performance Comparison. In *2014 International Conference on Computer, Communications, and Control Technology (I4CT)*, pages 324–328, Langkawi, Malaysia, September 2014. IEEE.

[KMT10] Maciej Kurant, Athina Markopoulou, and Patrick Thiran. On the Bias of BFS (Breadth First Search). In *2010 22nd International Teletraffic Congress (lTC 22)*, pages 1–8, September 2010.

[KS14] Michael Kirsche and Matti Schnurbusch. A New IEEE 802.15.4 Simulation Model for OMNeT++ / INET. In *Proceedings of the 1st International OMNeT++ Community Summit (OMNeT 2014)*, 2014.

[Laz13] Mihai T. Lazarescu. Design of a WSN Platform for Long-Term Environmental Monitoring for IoT Applications. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 3(1):45–54, March 2013.

[LCA12] Khawla Lahmar, Rym Cheour, and Mohamed Abid. Wireless Sensor Networks: Trends, Power Consumption and Simulators. In *2012 Sixth Asia Modelling Symposium*, pages 200–204, May 2012.

[LCH+11] Philip Levis, Thomas Heide Clausen, Jonathan Hui, Omprakash Gnawali, and JeongGil Ko. *The Trickle Algorithm*. RFC 6206. Internet Engineering Task Force, 2011.

[LH15] Jari Luomala and Ismo Hakala. Effects of Temperature and Humidity on Radio Signal Strength in Outdoor Wireless Sensor Networks. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 1247–1255, 2015.

[MHCK07] Gabriel Montenegro, Jonathan Hui, David Culler, and Nandakishore Kushalnagar. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. Internet Engineering Task Force, 2007.

[MPRS16] Ivan Minakov, Roberto Passerone, Alessandra Rizzardi, and Sabrina Sicari. A Comparative Study of Recent Wireless Sensor Network Simulators. *ACM Transactions on Sensor Networks*, 12(3):1–39, August 2016.

[MVK19] Levente Mészáros, Andras Varga, and Michael Kirsche. INET Framework. In Antonio Virdis and Michael Kirsche, editors, *Recent Advances in Network Simulation*, pages 55–106. Springer International Publishing, Cham, 2019.

[NHH18] Khoa Anh Ngo, Trong Thua Huynh, and De Thu Huynh. Simulation Wireless Sensor Networks in Castalia. In *Proceedings of the 2018 International Conference on Intelligent Information Technology - ICIIT 2018*, pages 39–44, Ha Noi, Viet Nam, 2018. ACM Press.

[ODE+06] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-Level Sensor Network Simulation with COOJA. In *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, pages 641–648, November 2006.

[OP12] George Oikonomou and Iain Phillips. Stateless Multicast Forwarding with RPL in 6LowPAN Sensor Networks. In *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 272–277, 2012.

[PBRD03] Charles Perkins, Elizabeth M. Belding-Royer, and Samir R. Das. *Ad hoc On-Demand Distance Vector (AODV) Routing*. RFC 3561. Internet Engineering Task Force, 2003.

[PK18] Nileshkumar R. Patel and Shishir Kumar. Wireless Sensor Networks' Challenges and Future Prospects. In *2018 International Conference on System Modeling & Advancement in Research Trends (SMART)*, pages 60–65, Moradabad, India, November 2018. IEEE.

[PR01] Charles E. Perkins and Elizabeth M. Royer. *Ad hoc On-Demand Distance Vector (AODV) Routing for IP Version 6*. Internet-Draft. Internet Engineering Task Force, 2001. `https://www.ietf.org/archive/id/draft-perkins-aodv6-01.txt` Accessed: 2022-02-26.

[RH10] George F. Riley and Thomas R. Henderson. The ns-3 Network Simulator. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 15–34. Springer, Berlin, Heidelberg, 2010.

[RK18] Philipp Raich and Wolfgang Kastner. The Need for Efficient Multicast Routing in Low-Power IPv6 Mesh Networks. In *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 1–4, Imperia, Italy, June 2018. IEEE.

[SCK+14] Niksa Skeledzija, Josip Cesic, Edin Koco, Vladimir Bachler, Hrvoje Nikola Vucemilo, and Hrvoje Dzapo. Smart Home Automation System for Energy Efficient Housing. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 166–171, Opatija, Croatia, May 2014. IEEE.

[SRR+19] José V. V. Sobral, Joel J. P. C. Rodrigues, Ricardo A. L. Rabêlo, Jalal Al-Muhtadi, and Valery Korotaev. Routing Protocols for Low Power and Lossy Networks in Internet of Things Applications. *Sensors*, 19(9), 2019.

[SRRAM19] José V.V. Sobral, Joel J.P.C. Rodrigues, Ricardo A.L. Rabêlo, and Jalal Al-Muhtadi. Multicast Improvement for LOADng in Internet of Things networks. *Measurement*, 148:106931, December 2019.

[TEN+11] Kok Seng Ting, Gee Keng Ee, Chee Kyun Ng, Nor Kamariah Noordin, and Borhanuddin Mohd. Ali. The Performance Evaluation of IEEE 802.11 against IEEE 802.15.4 with Low Transmission Power. In *The 17th Asia Pacific Conference on Communications*, pages 850–855, October 2011.

[UWM+15] M. U.Farooq, Muhammad Waseem, Sadia Mazhar, Anjum Khairi, and Talha Kamal. A Review on Internet of Things (IoT). *International Journal of Computer Applications*, 113(1):1–7, March 2015.

[VH08] András Varga and Rudolf Hornig. An Overview of the OMNeT++ Simulation Environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, pages 1–10, Marseille, France, March 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[VWC+20] Xavier Vilajosana, Thomas Watteyne, Tengfei Chang, Mališa Vučinić, Simon Duquennoy, and Pascal Thubert. IETF 6TiSCH: A Tutorial. *IEEE Communications Surveys & Tutorials*, 22(1):595–615, 2020.

[WNP11] He Wu, Sidharth Nabar, and Radha Poovendran. An Energy Framework for the Network Simulator 3 (NS-3). In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, Barcelona, Spain, 2011. ACM.

[WTB+12] Tim Winter, Pascal Thubert, Anders Brandt, Jonathan W. Hui, Richard Kelsey, Philip Levis, Kris Pister, Rene Struik, Jean-Philippe Vasseur, and Roger K. Alexander. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. Internet Engineering Task Force, 2012.