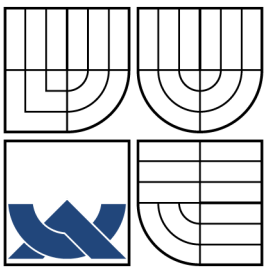
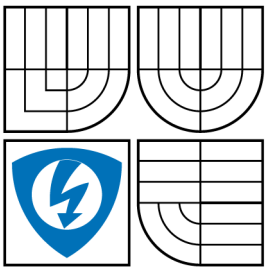


BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF RADIO ELECTRONICS

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV RADIOELEKTRONIKY

ROUTING PROTOCOLS FOR LOSSY WIRELESS NETWORKS

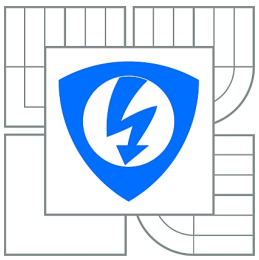
SMĚROVACÍ PROTOKOLY PRO ZTRÁTOVÉ BEZDRÁTOVÉ SÍTĚ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. ZENON KUDER

SUPERVISOR
VEDOUCÍ PRÁCE
doc. Ing. JIŘÍ ŠEBESTA, Ph.D.

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav radioelektroniky

Diplomová práce

magisterský navazující studijní obor
Elektronika a sdělovací technika

Student:	Bc. Zenon Kuder	***ID:***	106582
Ročník:	2	***Akademický rok:***	2011/2012

NÁZEV TÉMATU:

Směrovací protokoly pro ztrátové bezdrátové sítě

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se principy a řešením bezdrátových sítí typu mesh. Vytvořte model bezdrátové sítě typu mesh ve vhodném simulátoru sítě. Zhodnoťte vlastnosti sestaveného modelu vzhledem k již nasazeným sítím typu mesh společnosti Kamstrup. Implementujte protokol KMRF a RPL a proveďte testy použitelnosti RPL pro směrovací uzly napajené ze sítě. Testujte rovněž škálovatelnost RPL pro velké sítě.

DOPORUČENÁ LITERATURA:

[1] DAWSON-HAGGERTY, S., TAVAKOLI, A., CULLER, D. Hydro: A hybrid routing protocol for low-power and lossy networks. In proceedings of the 1st IEEE International Conference on Smart Grid Communications 2010. Gaithersburg: SmartGridComm 10., p. 268 –273, 2010.

[2] MULLIGAN, G. The 6lowpan architecture. In proceedings of the 4th workshop on embedded networked sensors. New York: EmNets 07, p. 78–82, 2007.

Termín zadání:	6.2.2012	***Termín odevzdání:***	10.8.2012

Vedoucí práce: doc. Ing. Jiří Šebesta, Ph.D.
Konzultanti diplomové práce:

prof. Dr. Ing. Zbyněk Raida
Předseda oborové rady

ABSTRACT

This thesis investigates suitability and design constraints of the NS-3 Simulator for simulations of wireless protocols used by Kamstrup metering infrastructure. An overview of NS-3 Simulator is given and preliminary implementations of two protocols are created. Wireless M-Bus as an example of a one-way protocol for battery-powered devices. The simulation of Wireless M-Bus is compared with measurements obtained in a real deployment. NS-3 proves to be a flexible framework for developing simulations of various network protocols, including the ones used for smart metering.

KEYWORDS

ns-3, simulation, wireless M-Bus, lossy mesh networks

ABSTRAKT

Tato práce zkoumá vhodnost a požadavky návrhu simulací pro simulátor NS-3 pro případ bezdrátových sítí používaných v měřící infrastruktuře společnosti Kamstrup. V práci je popsán simulátor NS-3 a je vytvořena základní implementace dvou protokolů. Wireless M-Bus jako příklad jednosměrného protokolu pro zařízení napájené z baterií. Simulace Wireless M-Bus je porovnána s daty naměřenými v reálném systému. NS-3 poskytuje flexibilní prostředí pro vývoj simulací různých síťových protokolů, včetně těch určených pro sítě inteligentních měřidel.

KLÍČOVÁ SLOVA

ns-3, simulace, wireless M-Bus, ztrátové sítě typu mesh

KUDER, Zenon *Routing Protocols for Lossy Wireless Networks*: master's thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Radio Electronics, 2012. 69 p. Supervised by doc. Ing. Jiří Šebesta, Ph.D.

DECLARATION

I declare that I have elaborated my master's thesis on the theme of "Routing Protocols for Lossy Wireless Networks" independently, under the supervision of the master's thesis supervisor and with the use of technical literature and other sources of information which are all quoted in the thesis and detailed in the list of literature at the end of the thesis.

As the author of the master's thesis I furthermore declare that, concerning the creation of this master's thesis, I have not infringed any copyright. In particular, I have not unlawfully encroached on anyone's personal copyright and I am fully aware of the consequences in the case of breaking Regulation § 11 and the following of the Copyright Act No 121/2000 Vol., including the possible consequences of criminal law resulted from Regulation § 152 of Criminal Act No 140/1961 Vol.

Brno ....

(author's signature)

Výzkum realizovaný v rámci této diplomové práce byl finančně podpořen projektem CZ.1.07/2.3.00/20.0007 **Wireless Communication Teams** operačního programu **Vzdělávání pro konkurenceschopnost**.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Finanční podpora byla poskytnuta Evropským sociálním fondem a státním rozpočtem České republiky.

ACKNOWLEDGEMENT

This project was developed at Aalborg University, Department of Electronic Systems, thanks to the Erasmus Lifelong Learning Programme.
The data and proprietary details of real electric meters deployments are courtesy of Kamstrup and the author would like to thank for that opportunity and cooperation.
Thanks to the supervisor at Aalborg University, Rasmus Melchior Jacobsen, for patience and support throughout the project. Additional thanks go to Jiří Šebesta from Department of Radio Electronics for his advice and helpfulness.

Brno ....

(author's signature)

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Wireless mesh networks are widely deployed in various environments and for different purposes. Kamstrup has been using a wireless mesh network technology in its widely deployed Advanced Metering Infrastructure (AMI). While a proprietary protocol called Kamstrup RF protocol (KMRF) has been used successfully in very large deployed networks, Kamstrup is in the process of upgrading the protocols used in order to support new applications.

A protocol improvement or replacement can improve the network parameters in several ways. For example extending the bandwidth and optimizing its use can allow more devices to be managed using the same resources, allowing e.g. improvement of smart grid applications. Additionally, in cases where power consumption is a concern, it is important to design the wireless protocol suite to fulfill the requirements, such as having the lowest possible power consumption or having the consumption predictable, so a device can have a guaranteed lifetime.

To be able to efficiently study the behaviour of the proposed mechanisms and to compare their performance, it is crucial to develop a simulation framework that allows the concepts to be tested with low cost, high flexibility and high level of realism, modelling the actual devices, their actual parameters and the radio transmission parameters of their deployments. In this project, the NS-3 simulator is chosen and used as the simulation tool. Two different network types used by Kamstrup are investigated: Wireless M-Bus (WM-Bus) and Kamstrup RF protocol 2.0 (KMRF2).

WM-Bus is a one-way data-gathering protocol for battery-powered heat and water meters. The main constraints are therefore power consumption and collision avoidance without the possibility to listen to the channel and get feedback from the network. Moreover, these devices can be placed in basements, therefore the signals often suffer from high levels of penetration losses. While the latter can be helped by adding repeater devices to the network, the former is solved by the communication protocol design, as described in section 3. The data broadcasted by the meters is gathered by devices called concentrators that are distributed around the target area.

KMRF, and its replacement, KMRF2, on the other hand, are more sophisticated protocols. They are used to gather information from electricity meters. The network consists of concentrators and meters. The concentrators are controlling a given set of meters and gather data and information about network topology from them. The concentrators use a source routing protocol to reach a given meter in the network, while the meters on the route forward the packet towards the destination. The routing decisions and the speed of propagation of information about link quality can have a strong impact on the overall performance of such a network.

The complexity of the protocols and especially the size of the network make it

unrealistic to test certain aspects of the protocols in real devices. Moreover, even if such a test was possible, tweaking the devices' firmware would require much more effort than editing the simulation models. The simulation therefore allows testing new design ideas very early in the development, while keeping the desired balance between the realism of the simulation and its complexity.

In this project, a simulation of a model of the WM-Bus procotol is created in the NS-3 simulation tool and simulated packet hit ratios are compared with actual measurements. A simple model of the KMRF2 protocol is then implemented while reusing parts of the WM-Bus model.

The scope of the simulation models can be extended to meet future needs, such as aid in planning the WM-Bus network or a test tool for further upgrades to the KMRF.

Chapter 2 gives an overview about the NS-3 simulator and its features, chapters 3 and 4 describe the WM-Bus and KMRF2 protocols and their models, respectively.

2 NS-3

NS-3 simulator was chosen as a versatile simulation tool that can be used to simulate different types of networks that are in use by Kamstrup including their potential upgrades and replacements. The core of NS-3 is a set of libraries that offers a framework and tools to support simulation development. Some of these tools are:

Scheduling – The core of the simulator. NS-3 is a discrete-event simulator, the scheduling is responsible for managing events.

Callbacks – Callbacks allow the simulation to stay flexible, because they allow calling methods without knowing their object's type.

Improved Objects – Objects derived from NS-3 `Object` class can use advanced features such as aggregation, easy downcasting and "smart" pointers, that count the number of references and automatically delete the object.

Attribute System – A system of string-based attributes for objects, that can be used to easily set different simulation and object parameters, from code and from the command line.

Command Line Arguments Support – Allows for easy addition of custom arguments and access to the attribute system.

Pseudo-Random Value Generator – Gives the same results each time the program is run unless set otherwise. The seed of the generator can be set from the command line using a *run number* that guarantees the pseudo-random streams will not overlap. This feature can be used to run multiple trials of the simulation, while keeping consistent results for each trial when desired.

Logging System – Offers several levels of verbosity that can be set per-model both from the main program and from the command line.

Unit Test Framework – In order to avoid introducing errors and to check existing code, tests can be prepared to check if the models behave in an expected way.

Tracing – To obtain results from the simulation, *trace sources* exist in the models and can be accessed by *trace sinks*.

In addition, the base package of NS-3 also contains existing models of several Internet protocols that can be used in simulations. Moreover, thanks to its open-source nature and popularity there are also models that are contributed or are in an early development. The models are not limited to just network protocols. There are, for example, models for mobility, propagation loss and energy consumption included.

NS-3 is written solely in C++ (as opposed to its predecessor, NS-2, which used a combination of C++ and OTcl) and offers binding for Python for most of its API [1] that can be used to create scripts, simulations or prototypes in Python.

The design of NS-3 also allows it to be run in real-time while connected to virtual machines running real networking stacks.

A short overview of a few other open-source network simulators is in the Appendix A. The NS-3 simulator was chosen as the most mature one with a potential for a stable future development.

2.1 Simulation Model Design

The NS-3 architecture is event-driven, i.e. the simulation, broadly speaking, consists of functions that schedule other functions to be run at a certain time. The simulation engine manages the proper order of the execution and offers a comprehensive framework for managing the execution and defining new simulation scenarios using the existing models.

NS-3 models are abstract representations of real-world objects, protocols, devices etc. [2] In general, these representations are implemented as classes, or sets of classes, with a given interface that are programmed to mimic certain aspects of the real objects. The choice of which aspects to simulate and how faithfully is a major concern when creating and using models. It is important to choose the right balance between the desired accuracy and complexity and speed. [1]

To achieve a high level of modularity and flexibility, callbacks, downcasting and object aggregation are supported by NS-3 and are widely used in the models. Callbacks allow functions to be passed as parameters and are often used to set up inter-layer data exchange. That approach helps modularity because one model does not need to keep a pointer to other model's object of a given type, it instead just keeps the callback reference. Aggregation of two objects allows the programmer to retrieve a pointer to one of the objects by using the pointer to the other. Downcasting allows to retrieve a pointer to a derived class by using a pointer to its base (parent) class. Both downcasting and getting the aggregated objects are done using the NS-3 method `GetObject()`.

In addition to the above, NS-3 defines a concept of a `Node`, an `Application`, a `Channel`, a `NetDevice` and Topology Helpers to aid developing simulations and implementing models [3].

A diagram of their relations is in the Figure 2.1. The diagram shows an example where there are two types of `Channel` used (e.g. a wired and a wireless channel) and therefore two types of `NetDevice` have to exist, as will be explained below. One `Node` has a `NetDevice` for both of them, the two others are only connected to one of the types. An `Application` can use any `Channel` depending on the address and its function. Topology Helpers are not included in the diagram.

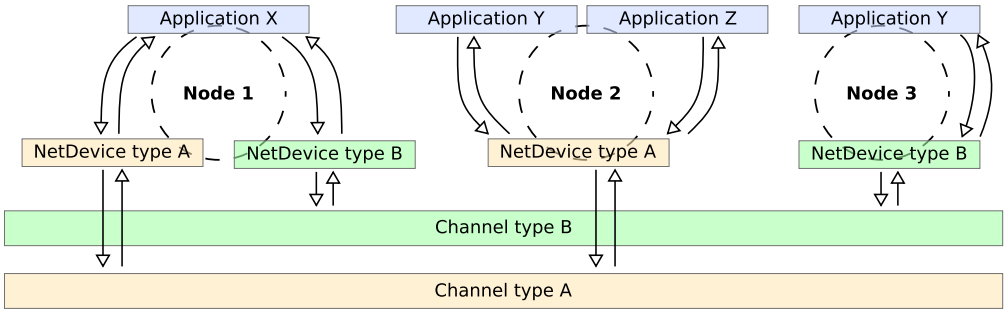


Fig. 2.1: Diagram of the basic blocks in NS-3 network simulations: `Node`, `Application`, `Channel` and `NetDevice`.

They are all representened by respective C++ classes and their meanings are:

`Node` represents a node in the network topology, or, more specifically, a computer or a device in the network. Devices and applications can be associated with a `Node` object in a similar way as in the real world and a `Node` keeps a list of pointers to all the associated `Application` and `NetDevice` objects. Additionally, thanks to the aggregation technique described above, `Node` objects can be associated with other objects, such as mobility model, that can be used to define the position and movement of a certain `Node`.

Using aggregation enables the simulation to stay flexible – an implementation of a certain model can check if an object of a given type is aggregated to its `Node` and be able to handle different scenarios. For example, a certain `Channel` model could be implemented to work both with `Node` objects that have their position set via a `Mobility` model, and with ones that do not.

`Application` class specifies the functionality for a `Node`. Its main purpose is therefore generating and replying to traffic.

To achieve this, the derived classes of `Application` may either create whole packets and exchange them directly with the lower layer models, or gradually build packets including their headers using a specific model for each layer. The choice depends on the desired realism of the simulation and design preference.

`NetDevice` represents a network interface and enables `Node` objects to communicate with each other over a given `Channel`. The `Channel` and the `NetDevice` have to be of a compatible type. In general, this class is used as an interface between the network layer and the device-specific functions, so the higher layer models can be switched and reused for different devices. This is especially useful for the Internet protocol suite.

Many existing protocol stacks in NS-3 introduce a separate `Phy` and `Mac` classes to model the respective layers when interfacing a `NetDevice` and a `Channel`.

`Channel` is an abstraction of the medium available to the node, typically a wire or

a wireless medium. One `Channel` is shared by several `NetDevice` objects of a compatible type and the `Channel` is responsible for keeping a list of connected devices and delivering data between them. There may be more `Channel` objects in the simulation and only the `Node` objects with a compatible `NetDevice` will be able to use them.

The `Channel` in this case does not represent a radio channel (meaning a frequency band), but rather a certain medium. If the model is expected to support several independent, non-interfering radio channels, the `NetDevice` and `Channel` implementation has to be adapted by introducing transmission channel numbers, central frequencies or other means to distinguish them. However, if it is useful for the implementation, it is possible to create a `NetDevice` that connects to multiple `Channel` objects, for example using two separate `Channel` objects for a Frequency Division Duplex (FDD) uplink and downlink. That approach is used e.g. in the current NS-3 LTE model [2], but becomes very inconvenient for a non-constant number of `Channel` objects.

Topology Helpers are classes designed to make the process of preparing the simulation easier, faster and more transparent. They are collections of methods that prepare groups of objects of the above types, configure them with given parameters and create the desired topologies and interconnections, so all the objects do not have to be created separately.

New models of `Application`, `NetDevice` and `Channel` are created by subclassing their respective classes, while `Node` is kept as an abstract concept defined by the associated objects. In contrast, the Helpers are created as stand-alone classes and they usually do not inherit from any of the NS-3 classes such as `Object`.

These classes do not represent a layering structure, but rather represent a skeleton the models may use and define general interfaces to encourage future compatibility. However, some of the defined interfaces are designed for the Internet protocol stack, and therefore are inappropriate for the models of the WM-Bus and KMRF protocols. An example of such interface is the `Send` method defined by the NS-3 `NetDevice` class, that requires a destination address and a protocol number as parameters. A protocol number is used in IP simulations to distinguish IP and ARP packets, and does not fit non-IP protocols. In case of WM-Bus, where all the transmissions are broadcast, the destination address is also unnecessary.

This problem can be avoided for example by passing meaningless values for the superfluous parameters and not using them, or by creating custom method with the desired functionality and defining the required methods to warn the user to use the custom method instead. Generally speaking, NS-3 does not require the model to adhere to the `Application`–`Node`–`NetDevice`–`Channel` structure at all. Nevertheless the models created in this project do adhere to it, but it is possible to

refactor them in the future, if necessary.

More details about the implementation are described in the sections for the particular protocols modelled.

2.2 Defining and Running a Simulation

2.2.1 Using the Helper Classes

The most straightforward way to define a simulation is to use the helper classes. An example procedure to do so is outlined in the Figure 2.2.

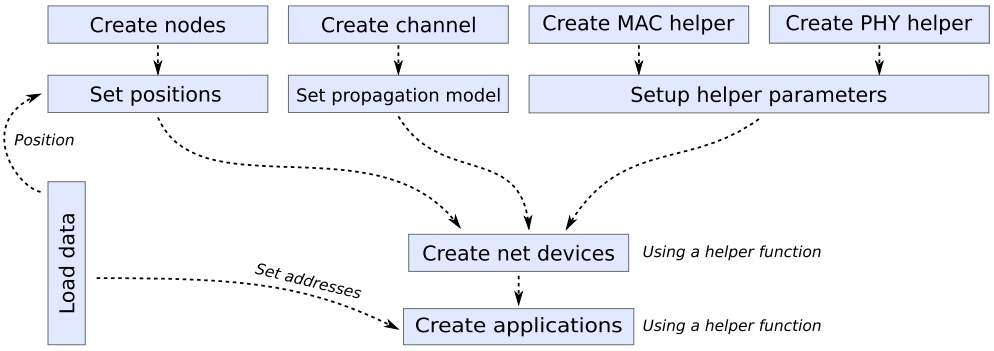


Fig. 2.2: Diagram of a simulation setup procedure using helper functions.

First, the desired number of `Node` objects is created, a `Channel` of the given type is created and, in our case, helper objects for the physical and the MAC layers. The helpers will later be used to create their respective objects with given attributes.

The `NetDevice` Helper uses the `Channel`, a `NodeContainer` containing the `Node` objects and both the helpers to create a new `NetDevice` object for each node. It uses the MAC and physical layer helpers to create new MAC and physical layer models and to connect them to the `Channel`. The application helper then creates a desired `Application` for each `Node`.

Loading the data and using it to set the address and the position, as outlined in the diagram, is specific for this project and has been developed for it, as it is not a usual part of NS-3 simulations. The tools for loading the measured data are described in Appendix B.

2.2.2 Using the Model APIs

If the helpers are not sufficient or not flexible enough, the users have the option to create all the objects and topology themselves. All the procedures that the helpers would do, such as creating the `Channel` and the `Node` objects, binding a `NetDevice`

with each `Node` and chaining the models for each layer of each `Node`, need to be implemented.

2.2.3 Creating a New Model

NS-3 already offers a wide range of existing models. When there is a feature that none of the existing models support, or a protocol that does not have a model yet, it becomes necessary to create a new model or customize an existing one.

As mentioned, NS-3 does not actually impose much limitations on how the models should be implemented. However, it does offer interfaces to adhere to and classes to inherit in order to save effort and make the model more useful. When creating a model, it is important to consider *functionality*, *reusability* and *dependencies*, as stated in the NS-3 Manual [1]. That being said, there are many choices of NS-3 classes a new model can inherit from, depending on the character of the future model.

- The `Object` class and its derived classes can use the NS-3 attribute system, the object aggregation and the smart-pointer counting system. That makes it the basic choice for most new classes. Also most of the classes in the NS-3 hierarchy inherit from it.
- When creating a new protocol model, it is useful to inherit from the `NetDevice`, `Channel` and `Application` classes in order to use an interface consistent with many existing models.
- To define a packet encapsulation structure, the classes can inherit from `Header` and `Trailer` to be able to easily add and read header and trailer fields from the packets. (The `Packet` class therefore does not have to be specialized, as all packets are built by adding headers and trailers).
- Addresses can be represented by the `Address` class. A new type of address is not created as a subclass of the `Address` class, but as a new base class. This new base class then implements methods `ConvertTo` and `ConvertFrom` that convert between the specific address type and an object of the generic `Address` class.
- There are existing models for propagation, error generation, energy consumption and other phenomena. The existing models are usually defined with a common parent class, therefore it is possible to create a new implementation and plug it into other models without changing them. An example of an inheritance hierarchy, showing the propagation loss parent class, is in the Figure 2.3.

The newly defined model has to be plugged into the simulation. In case it is a new model of a general type, such as a new propagation model, it can be used as a

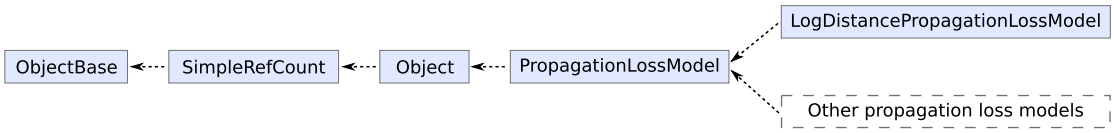


Fig. 2.3: A sample inheritance diagram showing a hierarchy of propagation models.

drop-in replacement for any other model of that type. In the less specific cases, the model has to be hooked into the simulation in some way, e.g. by setting a callback function of another model, changing the code of existing models or scheduling the new model's methods to be called when the simulation is started. If the new model is derived from the `Application` class, it should have defined functions that are called when the simulation runs.

2.3 Simple Wireless Module

Even though NS-3 contains a considerable amount of modules for different networking protocols, most of them are quite complex. While this is an advantage when using these modules, it becomes a disadvantage when attempting to modify them. In order to create a model of the protocols used in the current Kamstrup devices, a *Simple CSMA/CA Protocol for NS3*, further referred to as *Simple Wireless*, implementation by Junseok Kim [4] was used and modified. The model simulates WiFi and is implemented to allow easy modification.

It features the following classes:

- `SwChannel`, a derived class of `Channel`, as described above. Most of it has been reused in the models created in this project.
- `SwNetDevice`, a derived class of `NetDevice`. Parts of it were reused, parts were removed.
- `SwPhy`, a derived class of `Object`. This class models the physical layer of WiFi. Most of it was reused and expanded, the WiFi-specific parts were removed.
- `SwMacCsma`, derived of `SwMac` that is itself a derived class of `Object`. Models the MAC layer and most of it removed for use in the project, because there is no CSMA in the WM-Bus and the CSMA/CA in KMRF2 has not been implemented yet.
- Related helper functions to easily create a set of nodes equipped with the above.

3 WIRELESS M-BUS SIMULATION

To assess the feasibility and realism of the simulation, the first protocol to be implemented was Wireless M-Bus (WM-Bus) [5] mode C, that is used in some of the meters deployed by Kamstrup, e.g. heat and water meters. The main advantage of this protocol as a start of the project is the fact that the communication is only one-way, i.e. the meters broadcast their data, but do not receive any signals. Moreover, there were data available from a real deployment about positions of devices and measurements of received packets. It was therefore a good choice for making preliminary simulations and evaluating the performance of the propagation model.

3.1 Protocol and Network

The network consists of three types of nodes: meters, repeaters and concentrators. The meters periodically broadcast their state information. These signals are received by the concentrators, where the data can be processed. The repeaters are used in cases where the signals from the meters are not successfully received. They can be configured to re-broadcast signals received from given meters, effectively expanding the range of the concentrators in respect to those meters.

The Figure 3.1 shows a diagram of the configuration with two concentrators and one repeater. There are several meters in range of both concentrators – the meters are broadcasting their data, therefore both will receive them.

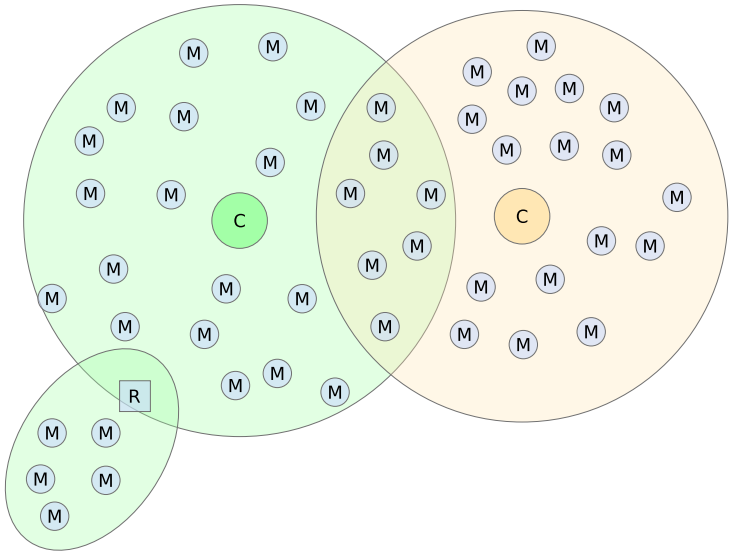


Fig. 3.1: Diagram of a Wireless M-Bus network with two concentrators (C), one repeater (R) and several meters (M).

To decrease the amount of redundant data sent, Kamstrup meters, such as *Multical 21*, *Multical 402* and *Multical 602* [6], use two types of frames: full and compact. The data in the compact frames can not be decoded if a full frame has not been previously received from that meter. Under normal conditions, a meter sends a full frame, followed by 7 compact frames and then repeats the cycle. However, in case there is an alarm occuring at the meter, the cycle is reset and a full frame is sent after the alarm, followed by 7 compact frames. The alarms can occur for various reasons, such as leak, burst, dry etc. The cycle is shown in Figure 3.2.

The alarm events occur independently at each meter and can be described as Poisson distributed.

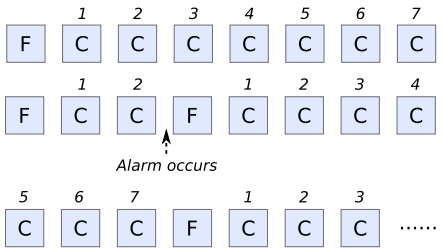


Fig. 3.2: Diagram of the full frame (F) – compact frame (C) cycle with one alarm event.

3.1.1 Protocol

The transmission frequency is around 868.95 MHz, using FSK at 100 kcps, the SINR required for successful decoding is assumed 8 dB for the repeaters and concentrators, as measured by Kamstrup, and the minimum receiver sensitivity for BER $< 10^{-2}$ according to the specification is -100 dBm.

The communication consists of a preamble and a synchronization sequence that is 64 bits long, followed by a link layer of 10 bytes and the data. The length of the whole frame can not exceed 256 bytes (not including the preamble and synchronization bytes).

To avoid collisions, the messages are transmitted periodically with a period changing according to the formula: [5]

$$T_{ACC} = (1 + (|ACC - 128| - 64)/2048) \times T_{nom}, \tag{3.1}$$

where $T_{nom} = N \times 2$ s.

- T_{ACC} – interval between two messages of a given device
- ACC – access number of the device
- T_{nom} – fixed nominal interval, a multiple of 2 seconds

- N – fixed unsigned integer. For Kamstrup meters $N = 8$, therefore $T_{nom} = 16$ s.

The access number ACC is a number specific to a given device and it is incremented and put to modulus 256 after each synchronous transmission. The initial value of ACC is chosen randomly for each device when deployed. Using delays calculated this way helps to avoid multiple subsequent collisions between given devices. This is achieved given the assumption that no two devices have started at the same time with the same access number.

All the communication happens at this defined intervals, i.e. when an alarm occurs, information about it is sent out during the next scheduled broadcast.

An example of the time differences between two transmitters that started at the same time with access numbers 0 and 50 is in Figure 3.3. When the difference is zero, the two signals collide. In this example the transmissions collide about twice every 256 transmissions, i.e. $256 \cdot 16 = 68$ minutes.

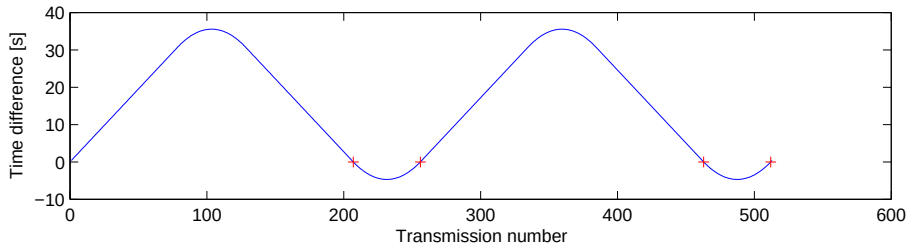


Fig. 3.3: Plot of transmitters started simultaneously with initial $ACC = 0$ and 50. Collisions occur when the difference is 0.

The transmission parameters are summarized in the Table 3.1.

Center Frequency	f	=	868.95 MHz
FSK frequency deviation			±45 kHz
FSK chip rate	f_{chip}	=	100 kcps
Data rate	$r_{data} = f_{chip}$	=	100 kbps
Speficied minimum receiver sensitivity	S	=	-100 dBm
Preamble length	$l_{preamble}$	=	8 bytes
Values valid for Kamstrup meters, e.g. Multical 602*:*			
Full frame length	l_{full}	=	89 bytes
Compact frame length	$l_{compact}$	=	62 bytes
Values valid for Kamstrup concentrators, using Texas Instruments CC1101 *chip:*			
Required SINR	$SINR_{min}$	=	8 dB

Tab. 3.1: Summary of the WM-Bus transmission parameters. [5] [6]

3.2 Implementation

3.2.1 Structure

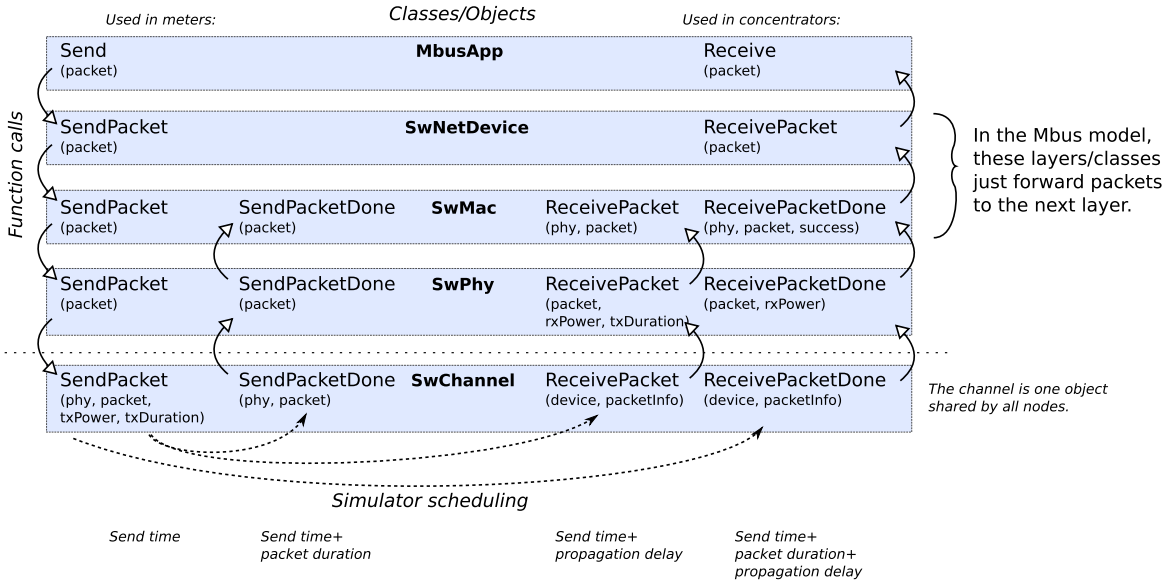


Fig. 3.4: Overview of the classes used in WM-Bus simulation.

As mentioned in section 2, the simulation in NS-3 offers a few basic classes to be used as a base structure for new models. The functionality of Wireless M-Bus protocol has been divided among them in the following way.

Application

The `Application` class has been subclassed into `MbusApp` class. This class is responsible for sending and receiving the full and compact messages and generating alarms. Each object takes care of its access number and counts its increments on every transmission.

As a design choice, one class is used for all nodes and the distinction between meters and concentrators is done through setting member variables of the objects. Setting up the objects is aided by helper classes `MbusBaseHelper` and `MbusNodeHelper`, that can create a set of objects of the same type given a list of coordinates and node addresses.

The frame payload is not relevant for the simulation, therefore the only content of the frame is the sender address in order to track which frames get delivered. The length of the frame, however, is remembered and used in the lower layers of the simulation to calculate the transmission duration.

Given the relative simplicity of the WM-Bus protocol, all the protocol-specific functionality has been implemented in the `MbusApp` class. Therefore the `SwNetDevice` and `SwMac` classes described below are mostly kept for structure and are just passing the packets to the next layer.

Net Device

A modified `SwNetDevice` class, subclass of `NetDevice`, from the *Simple Wireless* module is used. It forwards the packets from the `MbusApp` objects to the `SwMac` objects. Moreover, it keeps pointers to those objects and to the `Node` object, which can be useful for example to retrieve the node's coordinates from lower layers, or conversely, to retrieve a pointer to a node's lower layer given a pointer to the `Node` object.

There are `SwPhy` and `SwMac` classes. Most of their original functionality (used in the *Simple Wireless* Module) was removed, as the WM-Bus protocol is much simpler. Both of them are used at each node. The `SwPhy` commmunicates directly with the `Channel` object, forwards the data and most importantly, checks the power thresholds, sensitivity and SINR limits. The `SwMac` forwards the data and is kept in case a MAC layer model is desired in the future.

Channel

The `Channel` class has been subclassed into `SwChannel` class. One instance of this class is shared between all the devices and is responsible for dispatching the received data to all listening devices' `SwPhy` objects. It also calculates the interference levels for each device.

Events

As the NS-3 simulator is event based, it is necessary to implement the whole communication as a set of events. In the `Application` part of the simulation, the key events are the generation of messages and triggering the alarm.

The first message of a device is generated at a certain start time, that is either predefined or randomly chosen from a predefined range of delays (with uniform probability). While generating the first message, an event for the next message is scheduled with the delay defined by equation (3.1). Each message event schedules the following message at the appropriate time.

The alarm events are scheduled in a similar way, their delays are exponentially distributed with a configurable mean value.

In the lower layers, the key events are scheduled in the `Channel` instance:

1. Sending frame ends – scheduled when sending of the frame starts, the delay depends on the duration of the data.
2. Receiving frame starts – scheduled when sending of the frame starts as well, the delay is the propagation delay between the sending and receiving node.
3. Receiving frame ends – scheduled when receiving of the frame starts, the delay is the duration of the data.

Model of Frame Dropping

When the receiving functions are scheduled and run for each of the nodes, the simulation needs to decide whether the node was actually listening and whether the frame got successfully decoded given the signal level and interference.

This can be implemented in different ways depending on the level of realism required. In this project, the following approach is used, partially based on the behaviour of the *Simple Wireless* module.

The behaviour is illustrated in Figure 3.5. When a frame is received in the simu-

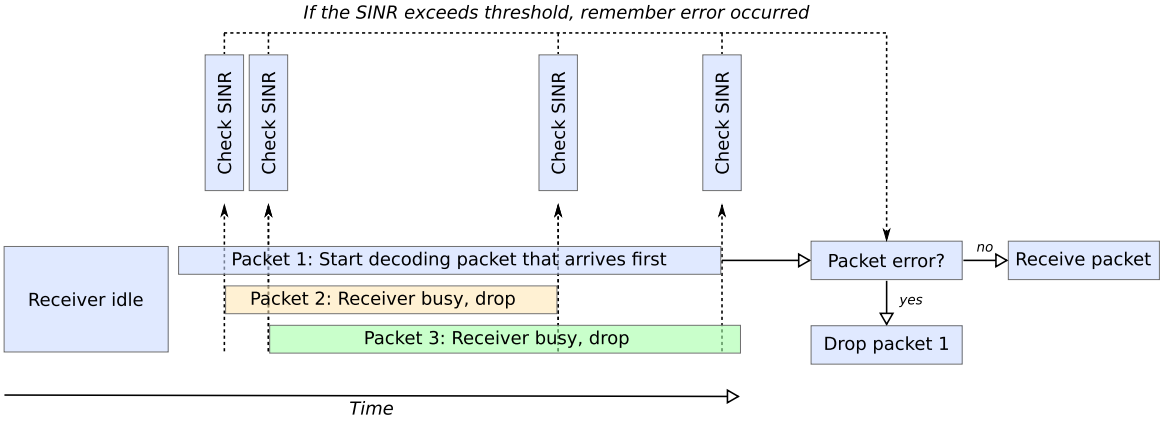


Fig. 3.5: Illustration of the frame dropping algorithm.

lation, its power is compared with the sensitivity of the receiver. If it is high enough, the frame is remembered and the state of the receiver is set to busy. Whenever any other message arrives, the SINR is calculated at the beginning and at the end of the reception to check if the SINR ratio of the frame being decoded stayed higher than the required value. In case the SINR gets too low at any point, an internal variable is set to indicate a packet error. The state of the variable is read after the reception of the desired packet and the packet is dropped in case there was an error. The thresholds are checked in `SwPhy::ReceivePacket` and `SwPhy::ReceivePacketDone`. If

the packet is being decoded, also `SwMac::ReceivePacket` and `SwMac::ReceivePacketDone` are called. The latter has an indication of success/failure that signals packet drop to the MAC layer.

The implementation in the *Simple Wireless* module checked the SINR only at the end of the reception, so, in terms of Figure 3.5, it would have missed the contribution of Packet 2 if it was shorter and ended before Packet 1.

The SINR is calculated as a ratio of the received desired signal power and the sum of all the other signals at the given receiver and the configurable noise floor, in linear values. The default value is set to $N_{\text{dB}} = S - SINR_{min} = -100$ dBm-8 dB $= -108$ dBm, assuming the specified sensitivity S (Table 3.1).

$$SINR_{dB} = 10 \log_{10} \left(\frac{P_{rx,\text{desired}}}{N + \sum P_{rx,\text{other}}} \right), \tag{3.2}$$

for the power in linear values of the desired signal $P_{rx,\text{desired}}$ and of all the other signals at the given device $P_{rx,\text{other}}$.

The realism of the frame reception could be improved by using a probabilistic model for dropping packets instead of the thresholding approach, e.g. by subclassing the NS-3 class `ErrorModel` and defining a packet error probability as a function of SINR.

3.3 Deployment Data

To make the simulation more useful and also to assess its performance, real data from Kamstrup deployment in Aabenraa were used. The tools for loading the data are described in Appendix B.

3.3.1 Geographical Data

A geographical dataset with the positions of the meters and repeaters and concentrators in KML format was available. Library `libkml` [7] was used to load the data. Additionally, it was necessary to convert the geographical coordinates (longitude and latitude) to Cartesian coordinates that could be used in the simulator and in the propagation models in general. The description of mapping the geographical coordinates to Cartesian coordinates is in the Appendix D.

3.3.2 Measured Values

Measurements from the concentrators were available as a set of SQLite databases. Among other information, they contained hourly values of: mean received signal

per a given node, the variance of that value and a number of successfully decoded messages.

After pairing the measurements with the geographical data, it was possible to export the nodes' distances and angles in respect to the concentrators and their orientation, as well as respective received power information. Such exported data could be used to assess different propagation models in other tools, e.g. Matlab.

A log-distance propagation model was chosen, because it is quite simple, yet can give satisfactory results and it has an implementation in NS-3.

3.3.3 Log-distance Propagation Model

The first and simplest model investigated was the log-distance path loss model, that can be defined by the following equation (in dB) [8]:

$$PL(d) = PL(d_0) + 10n \log(d/d_0) + X_\sigma \tag{3.3}$$

Where $PL(d)$ is the path loss at the distance d, $PL(d_0)$ is the known path loss at the reference distance d_0, usually the free space propagation loss at 1 m, 100 m or 1 km is used. The parameter n describes the relation between the distance and path loss and depends on the environment. X_σ is a zero mean Gaussian random variable (in dB) which represents the local shadowing that is assumed to be log-normally distributed.

However, while the direction and gain of the concentrators is known, there are less data about the orientation of the repeaters and meters, therefore omnidirectional gain pattern of 0 dB is assumed. There is even more uncertainty for the meters propagation losses, as the signal can be attenuated by penetration loss of walls and ceilings etc.

The propagation model was first tested using a Matlab script. In addition to the loaded data, an approximation of the concentrator antenna pattern was created based on its datasheet (Kathrein 800 10634 [9]). The maximum gain is assumed to be 16.5 dB and the horizontal pattern is approximated using a 12th degree polynomial that fits the following values in the least-squares sense:

angle [°]	0	30	60	90	120	130	140	150	170	180	190
angle [°]		−30	−60	−90	−120	−130	−140	−150	−170	−180	−190
gain [dB]	0	−3	−9.5	−18	−27	−30	−33	−30	−27	−26	−27

This yields the following polynomial (for x given in radians, using Matlab):

$$\begin{aligned} G = {} & 0.0007x^{12} + 0x^{11} - 0.0262x^{10} + 0x^9 \\ & + 0.3350x^8 + 0x^7 - 1.9155x^6 + 0x^5 \\ & + 5.4105x^4 + 0x^3 - 13.0066x^2 + 0x + 0.1508 \end{aligned} \tag{3.4}$$

The data were then used to find the appropriate parameters of the log-distance model. However, it turned out that the model fits better when the antenna pattern of the concentrators is assumed to be isotropic 0 dB gain.

Eventually, the simulations were run with $n = 2.97$, which was a result of fitting the data, and $PL(d_0) = 31.22$ dB, i.e. the free-space propagation loss at $d_0 = 1$ m for a frequency 868 MHz.

$$10 \log \left(\frac{4\pi \cdot 868.95 \cdot 10^6}{3 \cdot 10^8} \right)^2 = 31.22 \text{ dB}. \tag{3.5}$$

The model could be improved using knowledge about repeaters – they are mounted on light poles approx. 6 m high. On the other hand, there are not that many repeaters in the data set. Nevertheless, it remains a viable future improvement.

3.4 Simulation

Two types of simulation were run: one using the average path loss values from the measurements as a fixed path loss between the given meters and respective concentrators, and one using the log-distance propagation model to calculate the path loss. The simulation was set to log the sent and received frames into SQLite database in order to estimate the simulated hit rate, which was chosen as the parameter of interest.

3.4.1 Shadowing

A value modeling shadowing is added for each transmission in both cases – when using fixed path loss values from measurement and when estimating the path loss using the model. The value is taken as a realization of a zero-mean normally distributed random variable with a configurable standard deviation, in decibels, i.e. the linear value would be log-normally distributed. The packet hit results were the most similar to the measured ones when the standard deviation of this variable was 3 dB.

3.4.2 Detection Threshold

Another configurable parameter of the simulation, also used in both cases, is the sensitivity of the receiver, i.e. the minimum received power that results in a successful packet reception.

The value giving the best results was -100 dBm.

3.4.3 Results Comparison

Three types of data were compared – the measured path loss as given in the data set was used as a reference, simulation results with measured path loss and with simulated path loss were then compared with it, as outlined in Figure 3.6.

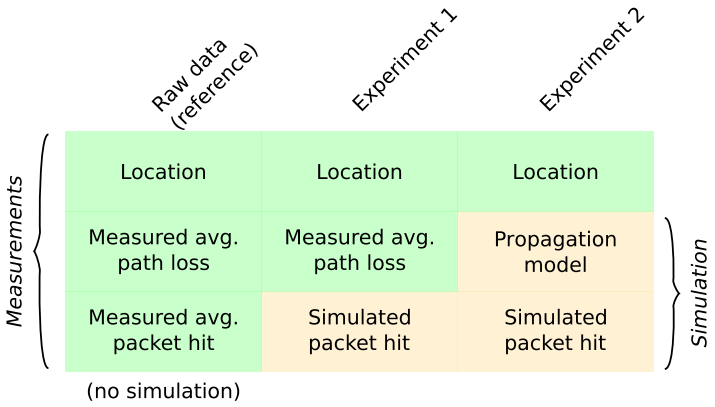


Fig. 3.6: Overview of the simulations.

The key point of interest was the packet hit rate value, which represents the number of packets from a particular meter received at a given concentrator in a unit of time, e.g. an hour. With a known number of packets sent, it can be represented as a percentage indicating the success rate of packet delivery:

$$r_{hit\%} = \frac{N_{received}}{N_{sent}}. \tag{3.6}$$

Using Measured Path Loss Values (Experiment 1)

The average measured values of the path loss between concentrator-meter pairs were extracted from the database. The values can be expressed in the following way:

$$P_{rx,avg;c,m} = \underset{t=1h,2h...}{\text{mean}} \left(P_{rx;c,m}(t) \right) \tag{3.7}$$

For all points where $P_{rx} \neq 0$, because zero values in the database were considered false positives or errors in the data. The average received power $P_{rx,avg}$ for a given concentrator c and meter m pair was calculated as an arithmetic mean of the average decibel values measured each hour by the concentrators (the hour-averages were in the available dataset).

Using Simulated Path Loss Values (Experiment 2)

The simulated values were calculated using the log-distance model as defined above in section 3.3.3, assuming a 10 dBm transmitters power.

Comparison with the Measured Hit Rate

The reference data was the measured hit rate, taken from the same dataset as the path loss in a similar way:

$$R_{avg;c,m} = \underset{t=1h,2h...}{\text{mean}} \Big(R_{c,m}(t) \Big) \tag{3.8}$$

where $R_{avg;c,m}$ represents the average received hit rate for a given concentrator and meter pair.

The prepared data were then analyzed in Matlab. Only the concentrator-meter pairs available in both the simulated and measured datasets were considered. The datasets differed because the system was not yet completely deployed, so the measurements were not complete, but the coordinates used for the simulation included the locations that were planned for the future.

Figures 3.7 and 3.8 compare the hit rate values from the measurements with the simulated hit rate using measured path loss and the simulated hit rate using simulated path loss values.

Figure 3.7 shows how many concentrator-meter pairs (Y axis) were there for a given range of hit rates. The plot is drawn using lines for clarity, however it represents a histogram with 40 bins. There is a high number of pairs that have very low hit rates. These are all the pairs that have very poor connection, but at least one packet got delivered due to fluctuations in the signal strength, so they appeared in the database. The other peak appears for high hit rates that are close to the maximum, creating the U-shaped plot.

The Figure 3.8 shows how the error of estimation between the measured average hit rate and the simulated ones was distributed. The number of occurences on the Y-axis represents the number of concentrator-meter pairs that differed by an amount given by the X-axis. The range centered at zero difference is the number of concentrator-meter pairs that differed the least. The left half of the plot are pairs that were "optimistic" in the simulation, because the simulated hit rate was higher than the measured one. The right half, on the other hand, are the values that were simulated with lower average hit rate.

The plots show that even the log-distance propagation model yields meaningful results, even though they obviously do not get as close as the simulation with known average path losses.

The maximum achievable hit rate in this case, given the average transmission period of 16 s, is $3600/16 = 225$.

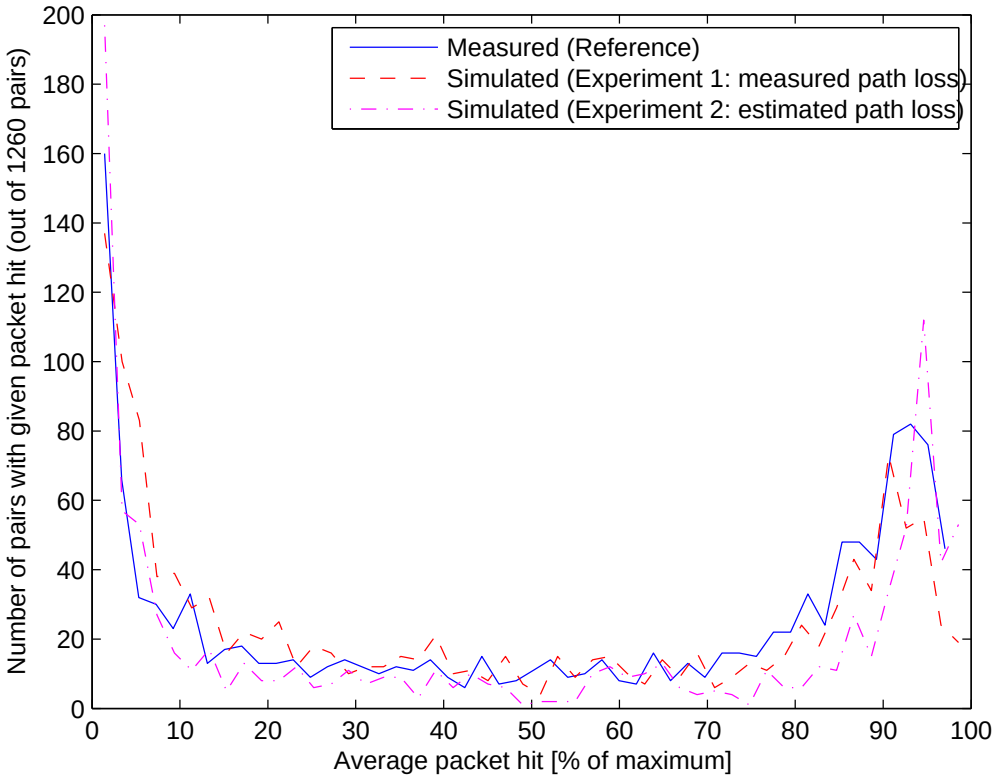


Fig. 3.7: Percentage (Y-axis) of concentrator-meter pairs (out of total 1260 pairs) being in the given hit rate range (X-axis).

3.5 Summary

Wireless M-Bus is a simple, one-way protocol, well-suited for simple battery-powered devices. The model implemented in this project is a coarse approximation of it, nevertheless it closely predicted the general packet delivery rate in a simulation based on data from a deployment in Aabenraa.

The model could be further improved by implementing a model of a repeater device. The predictions could be improved by using a more sophisticated propagation loss model, especially when more data about the installation of the meters is available, and by using a probabilistic model for dropping packets.

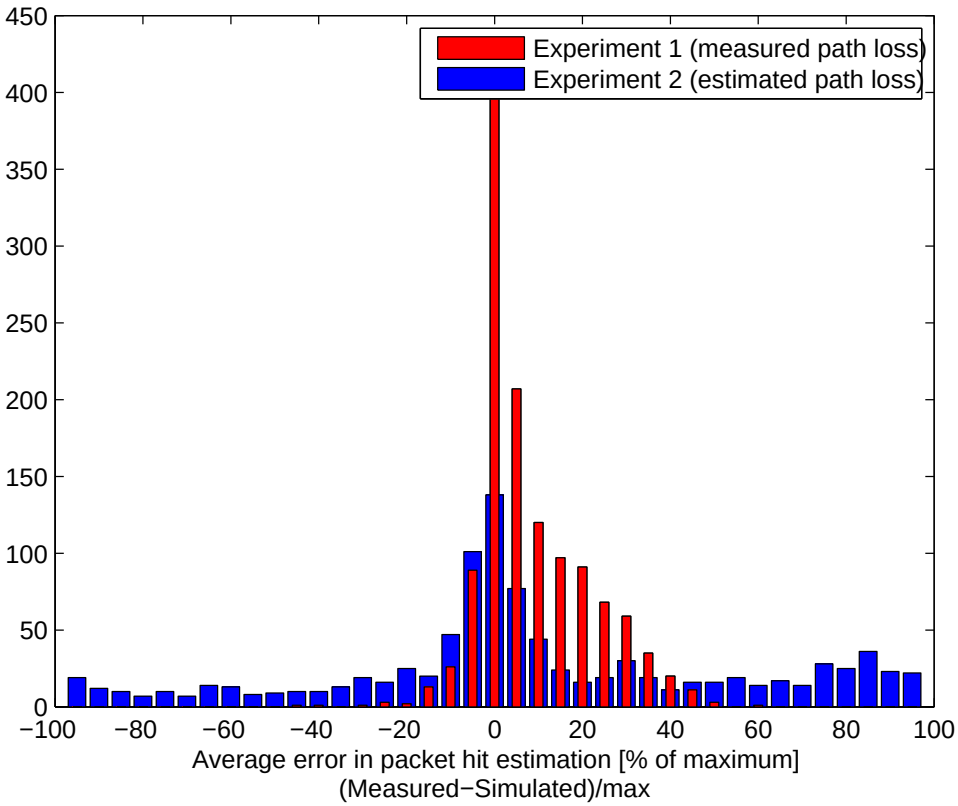


Fig. 3.8: Percentage (Y-axis) of concentrator-meter pairs (out of total 1260 pairs) where the packet hit estimation differs from the measurement by a given value (X-axis).

4 KAMSTRUP RF 2.0 SIMULATION

The main goal of the project is to simulate the current mesh network of electricity meters and to create a platform for testing impact of potential changes.

Considering the fact that Kamstrup is about to deploy a modified version of its current proprietary protocol, KMRF 2.0, the effort of this project concentrated on the new version. Therefore the NS-3 models created will be up-to-date and ready to be used for pre-development evaluation of different improvements to the current protocol, or, alternatively, to compare it with different protocols. Thanks to the popularity of NS-3, there are also third party models available for some of the standard protocols.

4.1 Network

The KMRF2 network consists of meters and concentrators. The concentrators' task is to keep an overview of the network in their range and to gather data from the meters. In addition, they provide network management such as firmware upgrades and time synchronization.

4.1.1 Network Lists

The concentrators keep the known meters in a *network list*. As the whole network consists of several concentrators and many meters, the concentrators are managed by system controllers that assign them encryption keys for the meters they are responsible for. Specifically, the meters in the network list can be in one of these three categories:

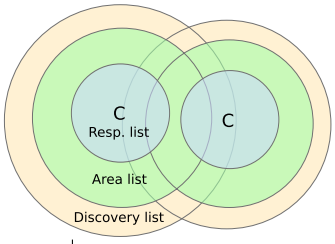
1. Responsibility list – List of the meters that the concentrator is responsible for.
2. Area list – List of the meters that the concentrator maintains a path to.
3. Discovery list – List of meters that were found in other meters' local (neighbour) list.

The Responsibility and Area lists are assigned by the controller, while the Discovery list is created by the concentrator from the data it receives.

The concentrators know the encryption keys of and keep routes to all meters in the Responsibility list and Area list. Any meter can only be in one concentrator's responsibility list, as shown in the diagram in Figure 4.1. In this diagram, the circles do not illustrate coverage, but represent the lists as sets with the following relations:

$$(R_i \subseteq A_i \subseteq D_i \subseteq M) \wedge (R_i \cap R_j, i \neq j) = \varnothing \tag{4.1}$$

for R_i, A_i, D_i being the i-th concentrator's responsibility, area and discovery lists, respectively, and M being all the meters in the network.

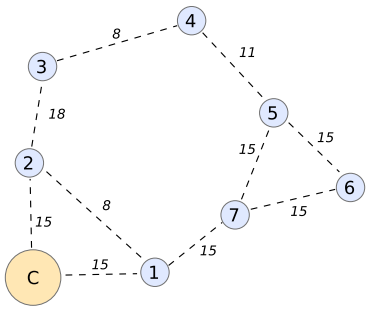


List	Aware of meter	Has encryption key	Maintains route
Responsibility	Y	Y	Y
Area	Y	Y	Y
Discovery	Y	–	–

Fig. 4.1: Diagram of two concentrators (C) and their lists.

4.1.2 Topology and Routing

The Figure 4.2 shows an example of the network topology with one concentrator and meters with numbers 1–7.



Meter	Neighbour	SNR	Meter	Neighbour	SNR
C	1	15	5	4	11
C	2	15	5	6	15
1	2	8	5	7	15
1	7	15	6	5	15
2	1	8	6	7	15
2	3	18	7	1	15
3	2	18	7	5	15
3	4	8	7	6	15
4	3	8			
4	5	11			

Fig. 4.2: Example of a KMRF 2.0 network topology, including a simplified concentrator local list table.

The concentrators can reach the meters using source routing. The whole route to the destination is set in the packet and the meters are able to read the route and

forward packets towards the destination. To successfully reach all the meters, the concentrators keep a list of the meters and their neighbours. A specified command is used to retrieve meters' *local list*, where their neighbours are stored. The meters' retrieved local lists are stored in the concentrator's local list table, as indicated in the Figure 4.2. The local list table is then used to build a routing table, as descried below.

The neighbours are discovered and saved in the local lists through beacon signals that every meter broadcasts. The beacon signals are sent out periodically at intervals randomly chosen from 14–15 minutes, except for startup when the interval is 5–15 minutes.

In addition to the discovered meters' addresses, the local lists store the last measured and the average signal-to-noise ratio and a hit-rate byte used to express the long-term stability of the link by storing which of the last 6 beacons were received and whether that particular neighbour has been in the list for at least 6×15 minutes.

Network Maintenance

The concentrators build the routing tables during the *network maintenance*. There are two types of network maintenance – *standard network maintenance* is run at least 5% of the time and includes retrieving the local lists of the meters before computing the routes. The *on-demand network maintenance* is run whenever a broken link is discovered and it recomputes the routes using the existing local lists.

The local list table of all the meters' neighbours is used to generate a weighted network graph that is used to calculate a route to each of the meters in the Area list. The links' weights are obtained from the measured SNR using the following look-up table:

$\gamma = \mathrm{SNR_{dB}}$ range (hex)	Weight (hex)	$\gamma = \mathrm{SNR_{dB}}$ range (hex)	Weight w (hex)
0x00 $\leq \gamma <$ 0x07	0x01	0x0F $\leq \gamma <$ 0x10	0xEC
0x07 $\leq \gamma <$ 0x08	0x36	0x10 $\leq \gamma <$ 0x11	0xF2
0x08 $\leq \gamma <$ 0x09	0x5D	0x11 $\leq \gamma <$ 0x12	0xF7
0x09 $\leq \gamma <$ 0x0A	0x7C	0x12 $\leq \gamma <$ 0x13	0xF9
0x0A $\leq \gamma <$ 0x0B	0x97	0x13 $\leq \gamma <$ 0x14	0xFB
0x0B $\leq \gamma <$ 0x0C	0xAD	0x14 $\leq \gamma <$ 0x15	0xFC
0x0C $\leq \gamma <$ 0x0D	0xC1	0x15 $\leq \gamma <$ 0x17	0xFD
0x0D $\leq \gamma <$ 0x0E	0xD3	0x17 $\leq \gamma <$ 0x19	0xFE
0x0E $\leq \gamma <$ 0x0F	0xE2	0x19 $\leq \gamma$	0xFF

The weight of the whole link is then calculated as (hexadecimal values, 0x100 =

256):

$$\text{weight} = 0\text{x}100 - \sum_i (0\text{x}100 - w_i) \tag{4.2}$$

where w_i is the weight of the i-th hop and the higher the weight, the better the link.

4.2 Protocol

The protocol follows a traditional layered structure.

4.2.1 Physical Layer

16 bytes	4 bytes	1 byte	1 byte	x bytes	2 bytes
Preamble	Sync word	Start of frame	Frame length	Payload	CRC

Before the frame starts a 16-bytes preamble and 4-bytes sync word are transmitted, followed by 1 byte start of frame (SOF) and 1 byte frame length. The bits are NRZ-encoded, the transmission speed is 4.8 kbit/s.

4.2.2 Data Link Layer

5 bytes	5 bytes	1 bytes	0-2 bytes	x bytes
Destination address	Source address	Frame control field	Transport time	Payload

Data link layer defines byte order, hop-to-hop transport, acknowledgments, addresses, reliable transport using retransmission, estimation of the transport time and the appropriate headers. The maximum frame length is 255 bytes and the payload is followed by 2 byte CRC.

The data link layer uses *full addresses*. The full address is 5 bytes long and contains a 3 bit network ID, 5 bit manufacturer ID and 32 bits serial number. Broadcast address is defined as the manufacturer ID and serial number set to ones.

The *frame control field* indicates whether an acknowledgment is expected and whether the *Transport time* field is set.

The data link layer transmissions are acknowledged. There are up to three retransmissions if the acknowledgment is not received. For each retransmission, a following CSMA scheme is used. Wait for the acknowledgment for 200 ms. If not received, wait additionally 61–100 ms and transmit if the channel is sensed clear. If the channel is blocked 3×300 ms, transmit after a total wait of 1200 ms.

4.2.3 Network Layer

1 byte	1 byte	1 byte	5 bytes	5 bytes	
Mode	Transfer ID	Control	Source addr.	Dest. addr.	
1 byte	1 byte	1 byte	1 byte	0-20 bytes	x bytes
Lowest path SNR	Hop limit	Number of addresses	Current address	Address list	Payload

Network layer defines the routing rules and headers for delivery between endpoints (i.e. concentrators and meters). The network layer also checks and avoids duplication of packets. The addresses used for endpoints are the full addresses described above.

The network layer header contains a *mode* field, that sets the mode to one of the following:

- 0x00 – *No header and no overhearing mode* – the network header does not contain any other fields and the application payload directly follows the *mode* byte. The packet is only processed if the link layer addresses matched the meter.
- 0x01 – *With header, no overhearing mode* – the default behaviour, the packet is passed to the application layer only if the network header destination matches the meter. Otherwise it is routed, if possible.
- 0x80 – *No header, overhearing enabled mode* – the network header does not contain any data, the payload is used as overheard data. The overheard packets do not generate acknowledgments. The link layer addresses do not have to match.
- 0x81 – *With header, overhearing enabled mode* – same as 0x80, however the network headers are processed first.

The network layer header contains the route as a list of the intermediate hops set by the concentrator. Only *short addresses* are stored in the list. A *short address* is a 16 bit address defined as the last 2 bytes of the device's serial number. The short address can be converted to a *full address* using a mask defined as `0x18 00 00 00 00` inserted between the net ID and the short address. A *full address* created using the mask is used in the data link layer, it can not be used as a network layer source nor destination.

There is also a byte *current address* with an index of the receiver address in the list. When forwarding a request from a concentrator, the current address is incremented and the address it points to is used as the next hop. When forwarding a reply to the concentrator, the current address is decremented.

The duplication of frames is avoided by setting a *transfer ID* field in the header. The transfer ID is incremented for each request sent by the concentrator. The meters

remember the last transfer ID and its direction and reject new packets with that transfer ID and direction.

To avoid loops in the path, the network layer headers contain a *hop limit* field that is set to 20 in the concentrator and is decremented at each hop.

When a delivery error in a request occurs, i.e. one of the hops does not receive a data link layer acknowledgment, the *network error* bit is set and the source address of the frame is set to the meter that did not receive the acknowledgment.

The network layer header also contains the lowest SNR of the reply path.

4.2.4 Application Layer

Application layer defines a set of commands that the meters understand, the expected data and header format for a request-reply communication. The application layer header contains, among others, a *command* byte, that defines what type of payload is going to follow.

There are several commands defined. In addition to reading the meter values (0x48), there are special commands for e.g. the beacons broadcasts (0x0A), for getting the local lists (0x53), for broadcasting the alarms (0xA1, 0xA3) for uploading new firmware (0xFC), etc.

4.3 Model Implementation

The NS-3 model created for this scenario extends the design used for WM-Bus, while reusing as much of the existing code as possible, particularly the channel and physical layer remained the same, the different parameters such as transmission speed can be configured from the helper functions using the attribute system.

Particularly, these parameters of the `SwPhy` objects were set using the attribute system to match the KMRF2 protocol:

Parameter	Value	Description
PreambleSize	$16 \cdot 8$	Preamble size (in chips): 16 bytes
SynchronizationSize	$4 \cdot 8$	Synchronization word (in chips): 4 bytes.
AdditionalPacketBytes	4	Bytes added to payload size: CRC: 2 bytes, frame length 1 byte, start of frame 1 byte
DataRate	4800	Bitrate: 4.8 kbit/s

4.3.1 Layers

Every layer described above was implemented as a separate class. For each node, instances of these classes are created and given pointers to the objects they need to communicate with. Helper functions were created to ease this process.

The diagram in the Figure 4.3 shows the relations between the objects and the methods used to exchange packets. The hierarchy does not include the `Kmrf2NetDevice`, as its functionality is limited mostly to keeping pointers to the `Kmrf2Mac`, `Kmrf2Phy` and `SwChannel` objects. Whether to not use a `NetDevice` in the implementation at all, or merge the functionlities of `Kmrf2NetDevice` and `Kmrf2Network`, i.e. the network layer implementation, needs to be considered.

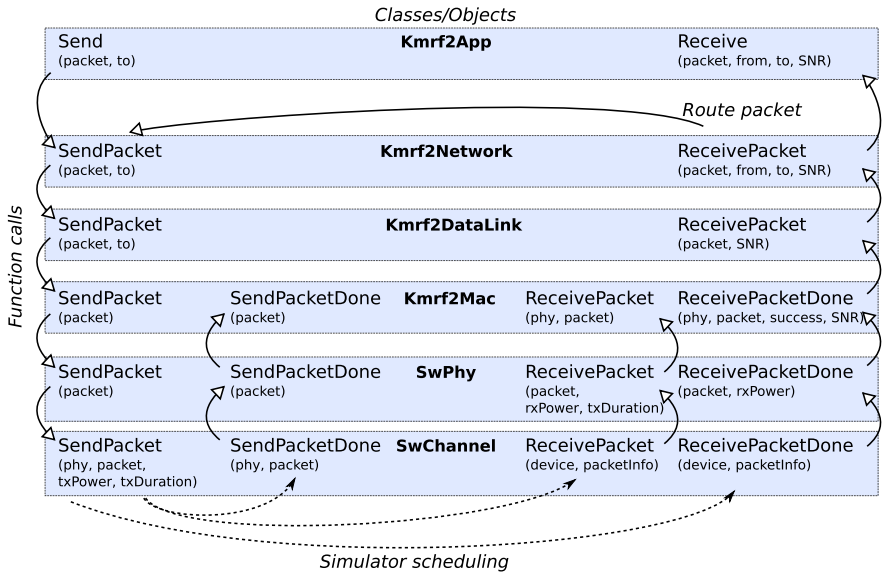


Fig. 4.3: Overview of the classes and the methods they use to communicate.

As shown in the diagram, the packets are processed and sent through all the layers. Additionally, the *SendPacketDone* and *ReceivePacketDone* methods are called at the channel, at the physical and at the MAC layer in order to simulate the time needed to send and receive packets and the related behaviour and state changes of the layers.

The `Application` class, `Kmrf2App`, has two subclasses: `Kmrf2MeterApp` and `Kmrf2ConcentratorApp`.

4.3.2 Addresses

KMRF 2.0 addresses, as described in section 4.2.2, can be of two types: *full address* (40 bits) and *short address* (16 bits).

As mentioned in section 2.2.3, any address can be represented using the extensible framework for creating address objects offered by NS-3. In this case, two classes were created – `Kmrf2AddressFull` and `Kmrf2AddressShort`. The former keeps separate member variables for the network ID, manufacturer ID and serial number, the latter only keeps the 16-bit serial number. They can be converted from and into a NS-3-defined objects of class `Address` that can be serialized and used in headers.

Thanks to this approach the addresses can be used with a common interface across the layers and their classes can be flexibly enhanced to suit the needs of the simulation. For example, in this project, the `Kmrf2AddressFull` class has been equipped with several constructors for different representations of the address. The full address object can also return a 64-bit integer representation of itself that can be used as an index of a list data structure. In a similar fashion, the `Kmrf2AddressShort` object has a method returning masked full addresses. Methods for comparing a unique full address with a masked one and testing whether the address is a broadcast were also implemented.

4.3.3 Headers and Packet Contents

NS-3 offers general `Packet` and `Header` classes. The `Packet` class represents a byte buffer and the `Header` class defines a few methods that have to be defined in the inherited classes to achieve compatibility with the `Packet` class, especially serialization and deserialization methods. In this manner, all the protocol headers and even the actual command payloads can be defined as subclasses of the `Header` class. Each of these classes has to define its serialization and deserialization methods that are used to translate the header data between serialized stream of bytes and object member variables. The complete packet is then built sequentially by creating the header objects and prepending them to the `Packet` object at the appropriate layers. NS-3 defines convenient methods of the `Packet` object for adding, reading and removing both headers and trailers. The `Packet` class is therefore used in its general form and there are no specific `Packet` subclasses used.

The model defines `Header` subclasses for the Application Layer, Network Layer and Data Link Layer and also for the application payloads used for the beacon command and for the local list retrieval command. A summary of the headers and the corresponding classes is in the Table 4.1. The implementation status is *yes* (y) in cases where the header value is used in the simulation. Some of the values marked as *not implemented* (–) are supported by the header classes, but are not used.

Layer header	Class name	Field	Type	Implemented?
Data link	Kmrf2DataLinkHeader	Destination	Kmrf2AddressFull (uint8_t[5])	y
		Source	Kmrf2AddressFull (uint8_t[5])	y
		Frame control field	uint8_t	–
		Transport time	uint16_t	–
Network	Kmrf2NetworkHeader	Mode		–
		Transfer ID		–
		Control	uint8_t	–
		Source	Kmrf2AddressFull (uint8_t[5])	y
		Destination	Kmrf2AddressFull (uint8_t[5])	y
		S/N	uint8_t	–
		Hop limit		–
		Number of addresses	uint8_t	y
		Current address	uint8_t	y
		Address list	std::vector <Kmrf2AddressShort>	y
Application	Kmrf2AppHeader	Req./Reply control		y
		Req./Reply ext. control		–
		Security header		–
		Command	uint8_t (enum)	y
		Clock		–
		Event status registers		–

Tab. 4.1: Layer headers, their corresponding classes and field types and implementation status.

4.3.4 Application and Network Layer Packet Dispatching

A global overview of the application and network functions used to build and deliver the packets is in the Figure 4.4.

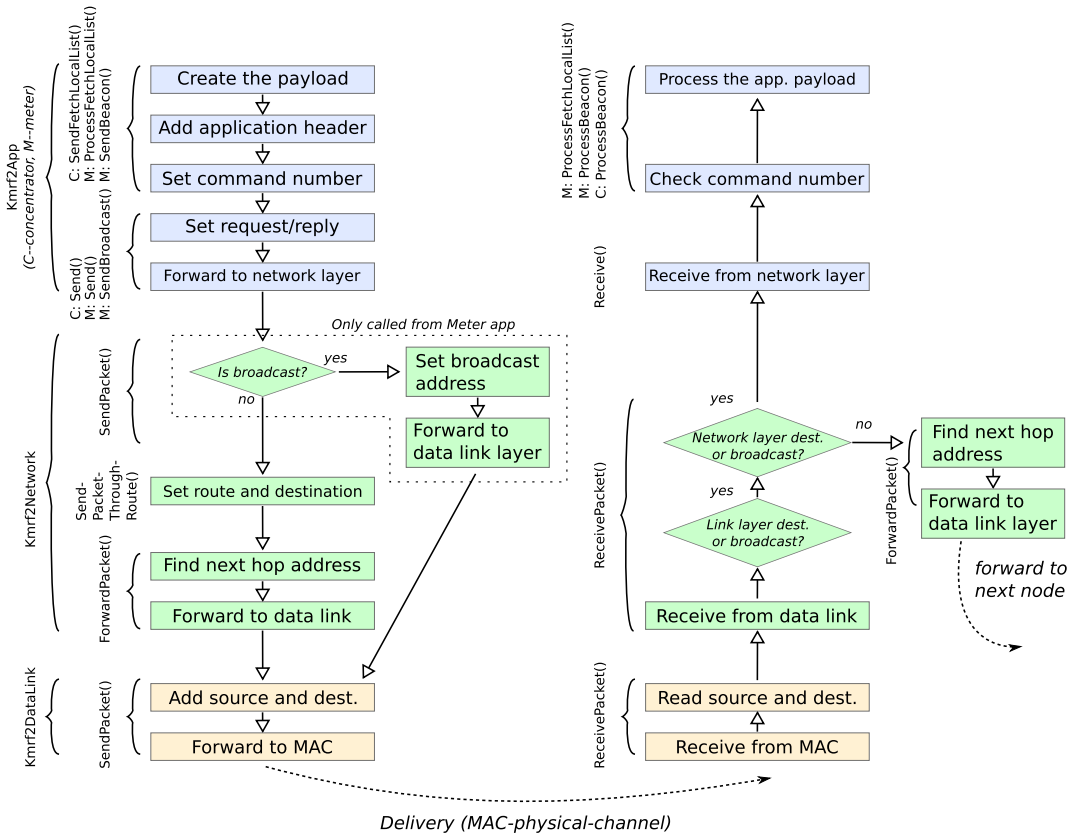


Fig. 4.4: Overview of the application and network layer functions participating in packet delivery.

Application Layer

The application layer supports several commands with different payload structures. When sending, the methods creating the payload are responsible for preparing the whole application layer packet, i.e. including their payload headers and the application header including the command number. The `Send` method then sets the request/reply to *request* in case of `Kmrf2ConcentratorApp` and to *reply* in case of `Kmrf2MeterApp` and forwards the packet to the network layer.

In concentrators, currently the only method that sends packets is `SendFetchLocalList()`, that is used to fetch a local list. In the meters, the packets are sent from `ProcessFetchLocalList()` as a reply to a query or from `SendBeacon()`, which uses a broadcast and therefore does not set the route.

When receiving, the application layer header is removed, the command number is read and the appropriate method is called to process the payload, such as `ProcessBeacon()` or `ProcessFetchLocalList()`.

Network Layer

When sending, the network layer sets the destination to the address defined by the application layer and the source to the address of the routing device. When initiating requests, i.e. calls from concentrator application layer, a method `SendPacketThroughRoute` is called to send a packet. The `Kmrf2ConcentratorApp` objects keep the routing table and pass the route as a parameter of this method.

Method `SendPacketThroughRoute` sets the route, the destination and sets the network layer reqest/reply bit according to the one set in the application header. The packet is then passed through method `ForwardPacket`, which increments/decrements the current address index in the route according to the reqest/reply bit and finds the next hop address. It then forwards the packet to the data link layer.

The `SendPacketThroughRoute` method is also called in meters when they are creating a reply. In that case, the route is set to the route that was remembered when the request arrived.

When a new packet is received from the data link layer, its *mode* byte is checked first. In case the mode is *with header, no overhearing* (0x01), the link layer destination is checked first. In case it is broadcast or it matches the meter's address, the network layer destination is checked. If it matches, the packet is sent to the application layer, otherwise it is forwarded using the `ForwardPacket` method. In case the link layer destination does not match, the packet is dropped.

For *no header and no overhearing* mode (0x00), only the link layer destination is checked. This, however, is not implemented in the current model.

4.3.5 Local Lists and Beacons

As mentioned in the section 4.1, the nodes listen to beacon signals from their neighbours and store their information in a *local list*, that can be retrieved by the concentrators during network maintenance.

When created, each of the nodes schedules the first beacon at a randomly chosen delay from the 5–15 minutes range specified by the protocol. The subsequent calls are scheduled each time the beacon is sent out for a randomly chosen delay of 14–15 minutes. This operation is shown in Figure 4.5.

The local lists in the simulation are stored as objects of a C++ class *Kmrf2LocalList* that keeps a `std::map` associative container of objects of class *Kmrf2LocalListItem*. The key of this container is the full address represented as 64-bit unsigned

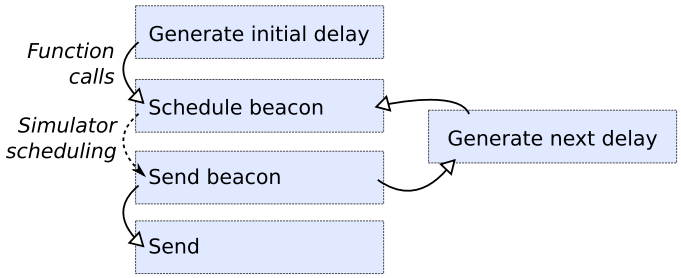


Fig. 4.5: Simple diagram of the beacon scheduling process.

integer.

In addition to data storage, the classes contain several methods for easy addition and updating of items.

The average SNR is calculated for each received beacon as an exponentially weighted moving average with 75 % gain ($w = 0.75$):

$$SNR_{avg,k+1} = w \cdot SNR_{new} + (1 - w) \cdot SNR_{avg,k} \tag{4.3}$$

where $SNR_{avg,k}$ is the average SNR after k received beacons. If there were no beacons previously received, it is set to the SNR_{new}, i.e. the SNR of the incoming beacon.

As defined by the protocol, the simulation also updates the *hit rate* field of the stored items every 15 minutes. There is a separate timer for that in each meter object, as the field needs to be updated even when the beacon is not received. The scheduling is implemented in a similar fashion as the beacon broadcast in the Figure 4.5.

Beacons use an application layer command 0x0A. The payload is quite simple and contains only 1 byte *command version*, 1 byte *meter SW version* and 2 bytes *meter type* fields. It is created as a header and prepended to a 0-length `Packet` object.

4.3.6 Network Maintenance

In the current implementation, the network maintenance is scheduled to run periodically, using similar scheme as sending the beacons, as described in section 4.3.5.

The implementation currently does not distinguish the responsibility, area and discovery lists. Each time the network maintenance is run, the concentrator sequentially fetches the local lists from all the meters in the *network list* (list of all the known meters, defined in section 4.1.1). In reality the concentrator would not fetch the local lists from the meters in the discovery list.

When the simulation is run, the network lists of the concentrators are empty. The concentrator therefore listens for the meters' beacons and adds the meters in range to its local lists table in addition to the items fetched from other meters.

When the network maintenance is run, the network list is first created from the unique items in the local lists table. Then, a routing graph is built from the local lists table and a routing table is calculated. Finally, the local list is fetched for every node in the network table.

The network maintenance overview diagram is in the Figure 4.6.

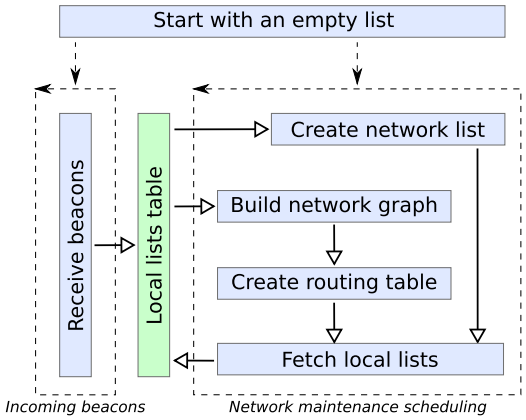


Fig. 4.6: Diagram of the network maintenance.

The implementation of fetching the local lists is illustrated in the Figure 4.7. For each item in the local list, a request is sent and a timeout is scheduled. If there is a reply before the timout runs, the timeout event is cancelled, the local list is stored and the procedure repeats for the next item in the network list.

The *Fetch local list* command uses the application layer command number 0x53 and is implemented as `Kmrf2AppFetchLocalListHeader` for the request payload and `Kmrf2AppFetchLocalListReplyHeader` for the reply including the neighbours list.

4.4 Model Test

Simple Test

The KMRF 2.0 model is in a very early stage of development. Only the basic parts of the protocol are implemented. However, receiving beacons, fetching the local lists and routing works. To test those features, a very simple test scenario has been created, as shown in the Figure 4.8. A concentrator and four nodes are distributed on a line in such way, that only the neighbours are in each others range, i.e. the concentrator can only communicate directly with meter 1, meter 1 can only

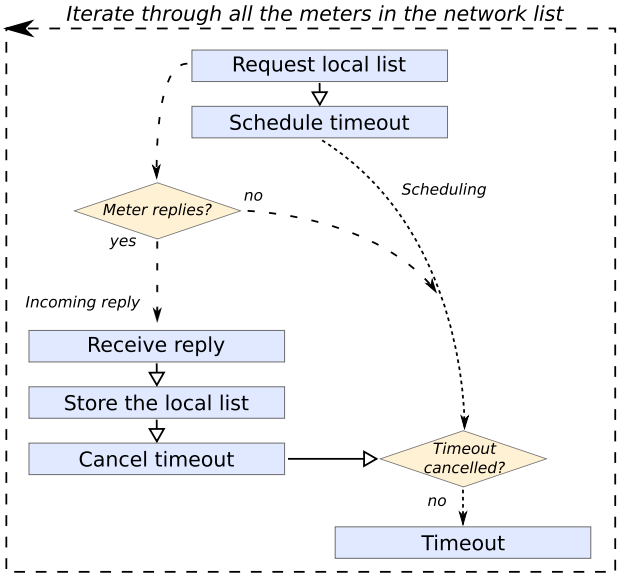


Fig. 4.7: Expanded diagram of *Fetch local lists* from the Figure 4.6.

communicate directly with the concentrator and meter 2, meter 2 only with meter 1 and 3, meter 3 only with 2 and 4 and meter 4 only with meter 3.

This was achieved by placing the nodes 100 m apart while using 0 dBm transmitter power, $n = 3.3$ path loss exponent for the log-distance model and -100 dBm sensitivity.

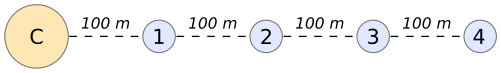


Fig. 4.8: A simple test scenario diagram.

The network maintenance was fixed to be run every 5 minutes, the beacons transmitted as defined by the protocol, i.e. first transmission after a random period from the uniform range 5–15 min, the subsequent periods 14–15 minutes. Only one cycle of the network maintenance was executed each 5 minutes, as opposed to the real protocol that runs the maintenance in a loop for a given amount of time.

Given that knowledge, the best and the worst expected number of network maintenance cycles before the meter 4 is discovered can be found:

NM count	Time since start [min]	Known meters after maintenance: Worst case	Best case
1	5	–	–
2	10	–	1, 2
3	15	1, 2	1, 2, 3
4	20	1, 2, 3	1, 2, 3, 4
5	25	1, 2, 3, 4	1, 2, 3, 4

The concentrator learns about meter 1 from its beacon. During the network maintenance it requests its local list, which should contain meter 2. In the following cycle, it requests the local list from both meter 1 and meter 2, thus learning about meter 3. This assumes the meters already received each others' beacons.

The difference between the worst and the best case is caused by the initial beacon delay of meter 1 and 2. If the concentrator gets beacon from meter 1 before the second network maintenance, and if the meter 1 is already aware of meter 2 by that time, both of the meters can be known after the second maintenance.

The table ignores the edge scenarios when the beacons would be broadcast at exactly 5 minutes or exactly 15 minutes.

The table indicates that except for the case when both meter 1 and meter 2 broadcast the beacon before 10 minutes after start-up, it should take the concentrator five network maintenance cycles to learn about all four meters. In reality, the network maintenance is run in a loop and therefore the concentrator in this scenario would learn about the meters right after all the beacons were delivered.

In 300 runs, using the NS-3 random numbers generator feature of creating new independent trials, the best case scenario, i.e. meter 4 in the concentrator's local lists table after 4 maintenance cycles, occured 80 times, i.e. $\approx 27\%$.

Test Including CSMA and Acknowledgments

After implementation of basic CSMA and acknowledgments, a new test was performed, using the positions of nodes from the WM-Bus data set.

In this case, the network maintenance repeats in loop for a certain period of time and rebuilds the routing table after finishing each cycle, as indicated in Figure 4.9. Between the network maintenance cycles data packets are sent. These data packets are not defined by the protocol, but are used to measure behaviour of the network in the simulation.

The results of this simulation are in Figures 4.10, 4.11, 4.12, 4.13, 4.14. Performance of the network with one and two concentrators was compared. The figures

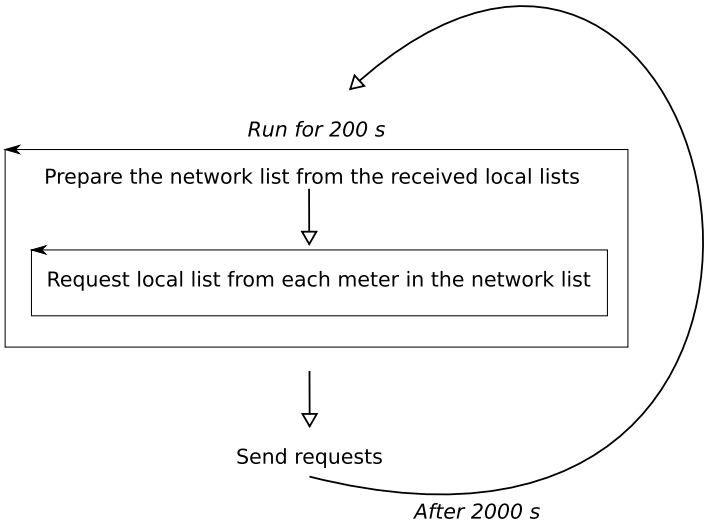


Fig. 4.9: Diagram of the network maintenance cycle.

show that there are many conflicts due to the fact that concentrators are quite close and try to communicate with the same meters, which would not happen in reality as this is controlled by the system.

The Figures 4.10 and 4.11 show how the concentrators sequentially learn about the meters when using the network maintenance algorithm described above. In reality, the algorithm would be more optimized and would not request the meters that are already known.

The Figures 4.12, 4.13 and 4.14 compare the packet loss, hop count and round trip time, respectively. They indicate that the implemented algorithms for retransmission work, as the round trip time for 2 concentrators is more spread out than the one for 1 concentrator.

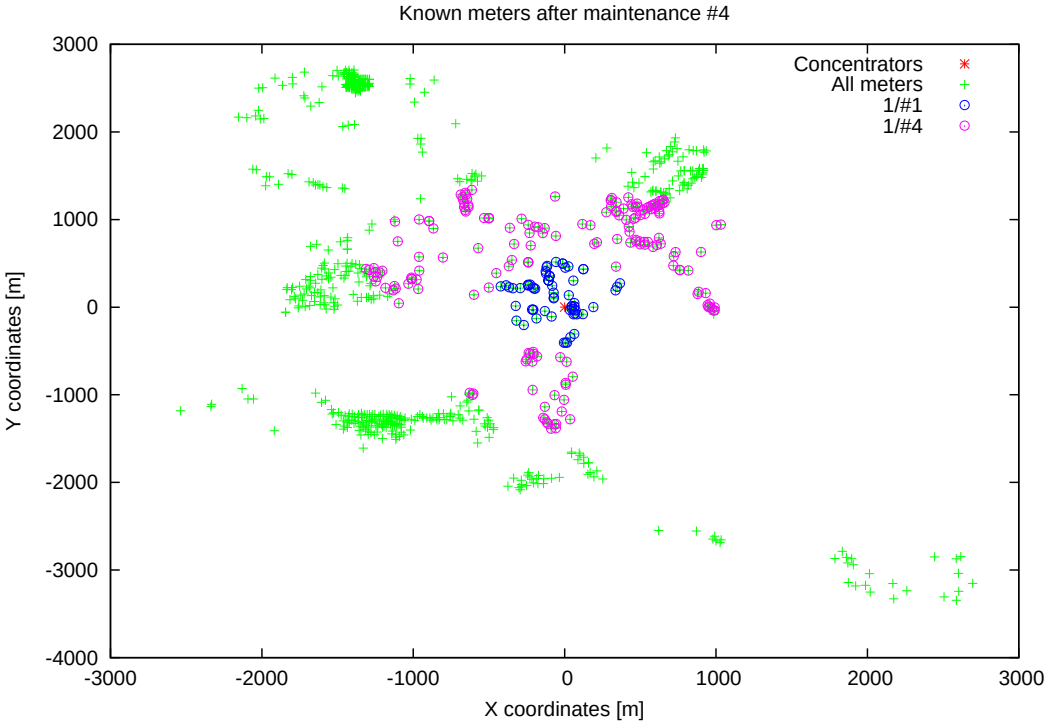


Fig. 4.10: Knowledge about the network – 1 concentrator.

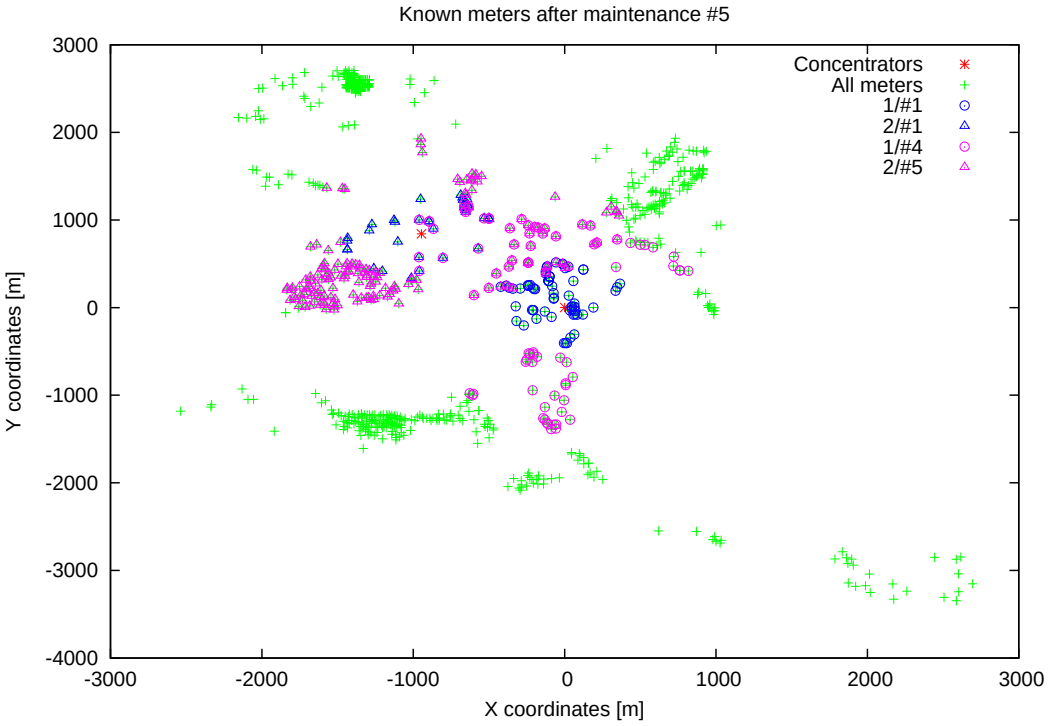


Fig. 4.11: Knowledge about the network – 2 concentrators.

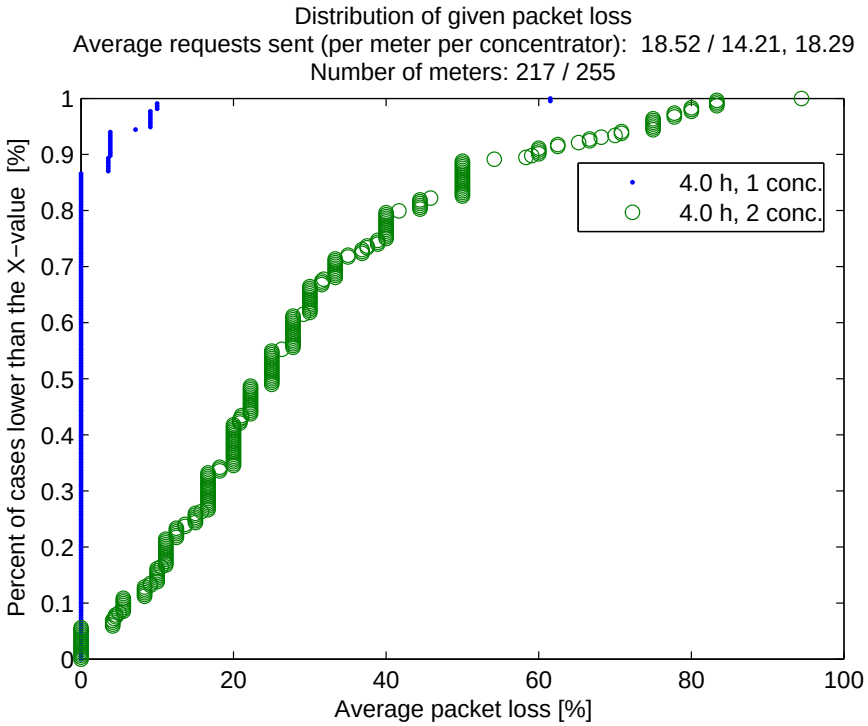


Fig. 4.12: Packet loss.

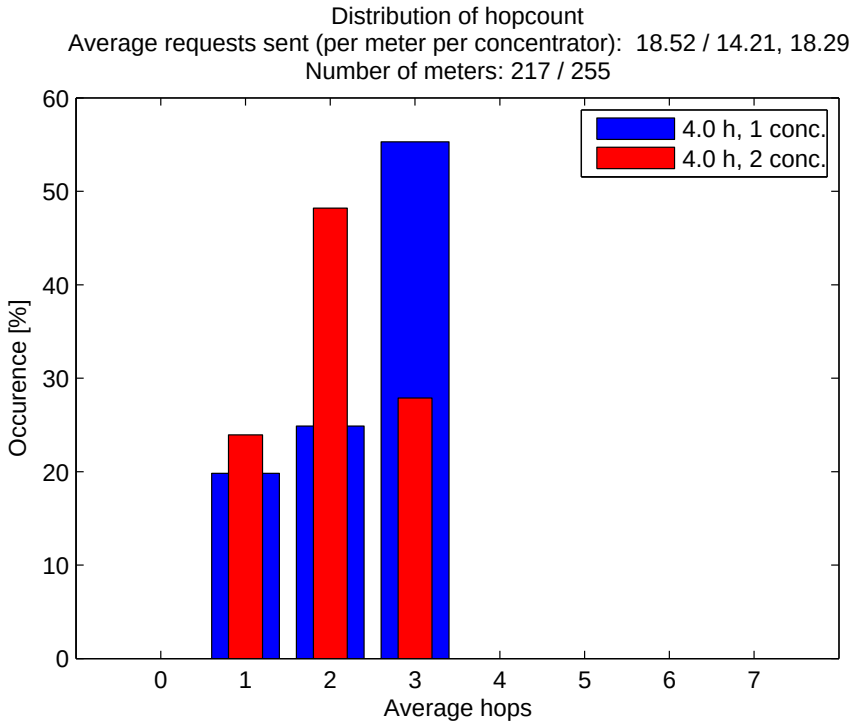


Fig. 4.13: Hop count.

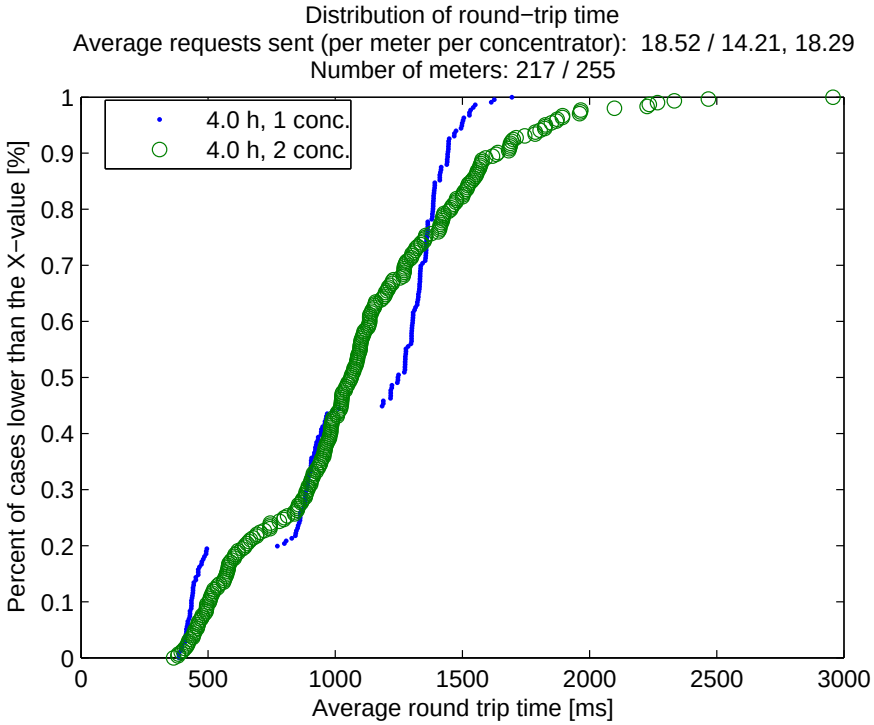


Fig. 4.14: Round trip time.

4.5 Summary

The Kamstrup RF 2.0 model created in this project is in an early stage. However its concept works as expected and proves that NS-3 is a flexible tool for simulation of different types of wireless networks.

After implementation of link error handling, specific application layer commands, advanced features of the protocol and realistic behaviour of the devices, it should be able to deliver results comparable with a real network. Further improvements could include a model of the controller, that would update the responsibility lists and request a certain amount of requests to be performed.

Given a working model, changes to the protocol could be investigated. For example, simulating the impact of concentrators sending multiple requests simultanously, instead of sequentially, on speed and reliability of the network. Or, additionally, seeing how much effect does the CSMA algorithm have. Checking how long does it take to learn about a network change, a broken link, or a whole network at start-up, and how do certain parameters of the protocol affect that.

5 RPL

IPv6 Routing Protocol for Low power and Lossy Networks (RPL) is a recently developed routing protocol. It was created as a reply to the demand of an IP-based routing protocol for lossy and low-power networks[10]. It therefore allows easy interoperability between these kinds of networks and the remaining IP-based Internet. Moreover, the use of IP allows easy reuse of existing code.

The routing in RPL is based on directed acyclic graphs (DAGs). It is a combination of distributed and centralized mechanisms. While the low-power nodes use distributed DAGs to find a default route to a *border router*, the border routers maintain a global view of the network topology[10].

RPL is of interest because it could serve as a replacement for other protocols used in low-power wireless networks, such as the ones described in this project. Due to time limitation, its model was not implemented as planned. However, there is an existing effort in the NS-3 community to develop such a model [11].

6 CONCLUSION

The NS-3 network simulator has been shown to be flexible and powerful enough to simulate two protocols of different types. Despite being simple in nature, the model for Wireless M-Bus was able to predict the general trend of hit rate in a real deployment. The implementation of the model for Kamstrup RF 2.0 is currently not very useful, however the main concepts of source routing, beacons, local list retrieval, Carrier Sense Multiple Access (CSMA) and retransmissions retrieval work.

In case of further upgrades, the models are easy to modify. Moreover, in case an upgrade to a standard protocol is considered, there might be existing models for that protocol, either directly in the NS-3 bundle, or as a third party contribution.

BIBLIOGRAPHY

[1] ns-3 project. (2012, Jan. 31). *ns-3 Manual, Release ns-3-dev* [Online]. Available: `http://www.nsnam.org/docs/release/3.13/manual/ns-3-manual.pdf`

[2] ns-3 project. (2012, Jan. 31). *ns-3 Model Library, Release ns-3-dev* [Online]. Available: `http://www.nsnam.org/docs/release/3.13/models/ns-3-model-library.pdf`

[3] ns-3 project. (2011, Dec. 23). *ns-3 Tutorial, Release ns-3.13* [Online]. Available: `http://www.nsnam.org/docs/release/3.13/tutorial/ns-3-tutorial.pdf`

[4] Kim, Junseok. (2011, Oct. 17). *Simple CSMA/CA Protocol for NS3* [Online]. Available: `http://www2.engr.arizona.edu/~junseok/simple_wireless.html` [Retrieved 2012-05-28].

[5] *Communication systems for meters and remote reading of meters – Part 4: Wireless meter readout (Radio meter reading for operation in SRD bands).* Draft prEN 13757-4:2011.10

[6] Kamstrup. *Mulical 21 Data Sheet.* `http://kamstrup.com/media/16541/file.pdf`.

[7] *libkml – a KML library written in C++ with bindings to other languages.* `http://code.google.com/p/libkml/` [Retrieved 2012-05-28].

[8] Jørgen Bach Andersen, Theodore S. Rappaport, Susumu Yoshida., "Propagation Measurements and Models for Wireless Communications Channels," *IEEE Commun. Mag.*, vol. 33, no. 1, pp. 42-49, Jan. 1995.

[9] Kathrein Scala Divison. *800 10634 65° Directional Antenna*, Datasheet [Online] `http://www.kathrein-scala.com/catalog/80010634.pdf` [Retrieved 2012-05-28].

[10] Dawson-Haggerty, S., Tavakoli, A., Culler, D. "Hydro: A hybrid routing protocol for low-power and lossy networks". In *proceedings of the 1st IEEE International Conference on Smart Grid Communications 2010.* Gaithersburg: SmartGridComm 10., p. 268 – 273, 2010.

[11] *Lr-wpan* in NS-3 Wiki [Online]. `http://www.nsnam.org/wiki/index.php/Lr-wpan` [Retrieved 2012-05-28].

[12] Lacage, Mathieu and Henderson, Thomas R. "Yet another network simulator" in *Proceeding from the 2006 workshop on ns-2: the IP network simulator.* ACM, New York, NY, USA, 2006

[13] *WSNet / Worldsens simulator homepage* [Online]. `http://wsnet.gforge.inria.fr/` [Retrieved 2012-05-28].

[14] *SQLite Home Page* [Online]. `http://www.sqlite.org/` [Retrieved 2012-05-28].

[15] NS-3 Wiki. *Installation* [Online]. `http://www.nsnam.org/wiki/index.php/Installation` [Retrieved 2012-02-13].

[16] John P. Snyder, *Map Projections: A Working Manual*, United States Government Printing Office, Washington, 1987.

[17] *Doxygen Home Page* [Online]. `http://www.stack.nl/~dimitri/doxygen/` [Retrieved 2012-05-30].

LIST OF SYMBOLS, PHYSICAL CONSTANTS AND ABBREVIATIONS

AMI	Advanced Metering Infrastructure
ARP	Address Resolution Protocol
BER	Bit Error Rate
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
FDD	Frequency Division Duplex
FSK	Frequency Shift Keying
IP	Internet Protocol
KML	Keyhole Markup Language (Used by Google Maps to store data)
KMRF	Kamstrup RF protocol
KMRF2	Kamstrup RF protocol 2.0
LTE	Long Term Evolution
MAC	Media Access Control
NRZ	Non-Return-to-Zero
RPL	IPv6 Routing Protocol for Low power and Lossy Networks
SINR	Signal to Interference and Noise Ratio
SNR	Signal to Noise Ratio
WM-Bus	Wireless M-Bus

LIST OF APPENDICES

A OTHER SIMULATORS

There are several simulation tools available, with different features, licenses and target uses. Several network simulators were considered: Omnet++, NS-2 and NS-3, WSNet and Contiki/Cooja.

A.1 Omnet++

Omnet++ is a modular network simulator with a wide range of modules with different features, including wireless sensor networks. It also includes a graphical interface integrated with Eclipse.

The network structure is defined using a special language called *NED*, while the library itself is based on C++.

However, the license only allows academic and non-profit use, therefore would limit the potential use at Kamstrup in the future.

A.2 NS-2

NS-2 used to be a very popular network simulator, with several projects expanding it for more specific tasks.

It is based on C++ and OTcl.

While still being maintained, it is not under very active development, because of its successor, NS-3.

A.3 NS-3

The first release of NS-3 was made in 2008. It is a completely new simulator, trying to overcome the problems of NS-2, especially the complicated use of two languages, some bad design decisions [12] and the emergence of several incompatible forked versions of it.

NS-3 is written in C++ and it has Python bindings to allow writing simple scripts in Python as well.

Even though NS-3 does not support as many protocols as NS-2 does, there is already a work in progress to implement 6LoWPAN and RPL [11], which are suitable for mesh networks.

A.4 WSNet

WSNet is a project specialized for large scale wireless networks [13]. The networks are defined by XML files, while the modules are defined in C.

However promising the features seem, the project also appears to be in early stage of developmnet and especially is lacking thorough documentation of the API.

A.5 Contiki/Cooja

Contiki is an operating system for wireless networks of low-power and low-memory devices. It includes a network simulator called Cooja, however it is designed to simulate nodes running the Contiki OS, therefore would be too specific for the purposes of this project.

B DESCRIPTION OF THE AVAILABLE DATA

B.1 Measurements

The measurements were available as a set of hourly SQLite datafiles for each of the concentrators, measured from 7 March 2012 to 27 March 2012 in Aabenraa.

There were 8 concentrators per one location: a separate concentrator for 2 polarizations in 4 directions.

Each of the files contained a table with a row for every meter and repeater whose signal was received during that hour. The columns this project used contained data about: meter number, meter type, average received signal, average hit rate.

To make the measurements more convenient, a small program was created in order to copy the contents of the datafiles into one master database.

The program's name is `crawllogs.cc` and it accepts a list of the datasets as arguments, the output database name is `outputdb.sqlite` and can be set in the source code. The program can be used using the following shell command to search for all datasets:

```
find <Datafiles location>/ -iname netlist.db -print0 | xargs -0 ./crawllogs
```

To increase the execution speed of the simulation and other operations, it was useful to also create a database with mean values only. A program `genMeansDB.cc` does exactly that – it loads a file `outpudb.sqlite` and generates a file `outputdb_means.sqlite`.

It was generated using the following SQL command, which represents the equation 3.7:

```
INSERT INTO allConcentrators
  (BaseNumber, Ip, Number, Manufacture, Type,
   SignalMean, SignalVariance, PacketHit, PacketHitRepeated)
SELECT BaseNumber, Ip, Number, Manufacture, Type,
  avg(SignalMean), avg(SignalVariance)/count(SignalVariance),
  avg(PacketHit), avg(PacketHitRepeated)
FROM orig.allConcentrators
WHERE SignalVariance!=0 AND SignalMean!=0 GROUP BY BaseNumber, Number;
```

Both crawllogs.cc and genMeansDB.cc can create their output databases, but leave the existing data if the database exists; in most cases it is therefore necessary to delete or rename the existing database files before running them. They both used the `libsqlite` [14] library to work with the SQLite files.

B.2 Positions of the Meters and the Concentrators

Positions of the meters were available as a KML file with the positions of the meters grouped by their availability in given concentrators, prepared for viewing in Google Earth.

Similarly, positions of the concentrators were available as a KMZ file, which is a compressed version of KML. Additonally, they were stored as lines starting at the concentrator position going in the direction of the antenna's orientation. The file was loaded using a function based on the `libkml` library [7].

Moreover, while the meter's KML file contained their serial numbers, the concentrators were only marked by their site and direction text. The concentrator names were assigned by parsing CSV data exported from a spreadsheet file that contained the site names and serial numbers.

The serial number, position and direction were saved into a list of structures with this information and used as an input of the Helper functions when setting up the topologies.

It would be useful to separate the parsing from the simulation code by saving the coordinates and serial numbers into a database, so only the database would be accessed from the simulation and the parsing functions could be independent.

C INSTALLATION AND USAGE

C.1 Installation on Fedora Linux

The following procedure describes how to install NS-3 on Fedora Linux 16 [15]. Other systems may differ, but the general concept should be similar.

```
~ # yum install gcc gcc-c++ python python-devel mercurial \
bzr gsl gsl-devel gtk2 gtk2-devel gdb valgrind doxygen \
graphviz ImageMagick python-sphinx dia flex bison \
compat-gcc-34 tcpdump sqlite sqlite-devel libxml2 \
libxml2-devel uncrustify libkml libkml-devel boost \
boost-devel uriparser uriparser-devel

[for x64 systems]
~ # echo "/usr/lib64/libkml" > \ /etc/ld.so.conf.d/kml-x86_64.conf
[for i686 systems]
~ # echo "/usr/lib/libkml" > \ /etc/ld.so.conf.d/kml-i686.conf
~ # ldconfig

~ $ mkdir repos
~ $ cd repos/
repos $  hg clone http://code.nsnam.org/ns-3-allinone
repos $ cd ns-3-allinone/
ns-3-allinone $ ls
build.py  constants.py  dist.py  download.py  README  util.py
ns-3-allinone $ ./download.py -n ns-3.13
ns-3-allinone $ ./build.py
ns-3-allinone $ cd ns-3.13
ns-3.13 $ ./waf -d debug --enable-examples --enable-tests configure
ns-3.13 $ ./waf
ns-3.13 $ ./test.py -c core
ns-3.13 $ ./waf --run hello-simulator
```

After NS-3 is successfully installed, the `sw` directory from the attachment E can be copied into the `ns-3.13/src/` directory:

```
$ cp sw ~/repos/ns-3-allinone/ns-3.13/src/
```

And the build can be reconfigured:

```
ns-3.13 $ ./waf -d debug --enable-examples --enable-tests configure
```

In case of problems with the linker, there are additional libraries set in `src/-sw/wscript` and their paths may need updating, especially on non-64-bit systems the paths need replacing `/usr/lib64` to `/usr/lib`.

C.2 Running the Simulations

Before running the simulations, the following paths to data files need to be set: `IPMAPPINGFILENAME` and `NAMEMAPPINGFILENAME` in `sw/model/ConcentratorMappingUtils.h` and `MEASUREMENTS_DB_NAME`, `METERS_KML_FILENAME` and `CONCENTRATORS_KML_FILENAME` in `sw/examples/mbus-simple-ex.cc`. Sample data are in the `Data` directory on the attached CD.

Then the simulations can be run. The WM-Bus simulation can be run using a command like this:

```
ns-3.13 $ ./waf --run "mbus-simple-ex"
```

Or, with parameters and output:

```
ns-3.13 $ ./waf --run "mbus-simple-ex --txPowerDbm=10 \
--pathlossExponent=2.97 --useMeasuredPathloss=0 --noiseFloorDbm=-108 \
--sinrThreshold=8 --shadowStd=3" 1> /dev/null 2>tmp.txt ; \
echo $STARTDATE; date; ./latestTrace2csv > outputPacketHits.csv; date;
```

And the KMRF2 simulation can be run with a command like this:

```
ns-3.13 $ STARTDATE=`date`; NS_GLOBAL_VALUE="RngRun=1" \
 ./waf --run "kmrf2-ex" ; \
echo $STARTDATE; date;
```

To run the tests of parts of the KMRF2 model:

```
ns-3.13 $ ./waf --run "test-runner --suite=kmrf2 --verbose"
```

D CONVERTING GEOGRAPHICAL COORDINATES

The positions of the meters, repeaters and concentrators were given in longitude and latitue, but to be practical in the simulation, they had to be converted to a Cartesian coordinate system. There are many map projections that could be used to achieve this goal. For the purpose of a rather small area of one city, the Universal Transverse Mercator coordinate system seemed to be a reasonable choice. This projection splits the area of Earth into 60 zones, 6 degress of longitude wide, and covers each of them using the Mercator projection.

To calculate the coordinates, the following equations can be used [16] – x increases east and y increases north. Given latitude ϕ, longitude λ and reference meridian longitude λ_0, most of Denmark is in zone 32 with $\lambda_0 = 9°$

$$x = x_0 + k_0 N \left(A + (1 - T + C)\frac{A^3}{6} + (5 - 18T + T^2 + 72C - 58e'^2)\frac{A^5}{120} \right) \tag{D.1}$$

$$\begin{aligned} y = y_0 + k_0 \left[M - M_0 + N \tan\phi \left(\frac{A^2}{2} + (5 - T + 9C + 4C^2)\frac{A^4}{24} + \right.\right. \\ \left.\left. + (61 - 58T + T^2 + 600C - 330e'^2)\frac{A^6}{720} \right) \right] \end{aligned} \tag{D.2}$$

where $k_0 = 0.9996$ for the UTM projection and:

$$e'^2 = \frac{e^2}{1 - e^2} \tag{D.3}$$

$$N = \frac{a}{\sqrt{1 - e^2 \sin^2\phi}} \tag{D.4}$$

$$T = \tan^2\phi \tag{D.5}$$

$$C = e'^2 \cos^2\phi \tag{D.6}$$

$$A = (\lambda - \lambda_0)\cos\phi \tag{D.7}$$

$$\begin{aligned} M = a \left[\left(1 - \frac{e^2}{4} - \frac{3e^4}{64} - \frac{5e^6}{256} - \dots \right)\phi - \left(\frac{3e^2}{8} + \frac{3e^4}{32} + \frac{45e^6}{1024} + \dots \right)\sin 2\phi + \right. \\ \left. + \left(\frac{15e^4}{256} + \frac{45e^6}{1024} + \dots \right)\sin 4\phi - \left(\frac{35e^6}{3072 + \dots} \right)\sin 6\phi \right] \end{aligned} \tag{D.8}$$

$M_0 = 0$, as it is the value of M at the latitude where the coordinate system begins, i.e. the equator ($\phi = 0$).

Eccentricity of the Earth ellipsoid	e	=	0.081892
Equatorial radius	a	=	6378.137 km
Point of origin	x_0	=	500 km
Point of origin	y_0	=	0 m (Northern hemisphere)
Point of origin	y_0	=	10 000 km (Southern hemisphere)

In all equations, λ, λ_0, ϕ are in radians. The origin values x_0 and y_0 are used to avoid negative numbers.

E ATTACHED CD

The contents of the CD are the following source files:

E.1 Source Code Documentation

An automatically-generated documentation of the source code is attached as a separate document. The documentation was generated using Doxygen [17]. The filename is `documentation.pdf`.

E.2 NS-3 Files

NS-3 files are contained in the directory `sw`. It contains following files:

`sw/`:

- **`wscript`** – File used by the NS-3 build system that defines all the files necessary for build.
- **`USAGE`** – Describes a few examples how to run a simulation.

`sw/examples/`:

- **`examples/kmrf2-ex.cc`** – The main simulation script for KMRF2 simulation. It defines an examples scenario as defined in 4.4.
- **`examples/mbus-simple-ex.cc`** – The main simulation script for WM-Bus. Can be configured to run the simulation or output the measured data for other scripts.
- **`examples/sw-linear-multihop-ex.cc`** – An example from the original *Simple Wireless* module.
- **`examples/wscript`** – Build script defining these applications can be run.

`sw/helper/`:

- **`helper/kmrf2-helper.h, .cc`** – Implementation of the main helper classes for KMRF2.
- **`helper/mbus-helper.h, .cc`** – Implementation of the main helper classes for WM-Bus.
- **`helper/kmrf2-mac-helper.h, .cc`** – Implementation of the helper class for creating `Kmrf2Mac` objects. Based on *Simple Wireless* module.
- **`helper/w-helper.h, .cc`** – Implementation of a helper class for binding the lower layers with the `SwNetDevice` and `Node` objects. From *Simple Wireless* module.
- **`helper/sw-mac-csma-helper.h, .cc`** – Helper class for `SwMacCsma` objects. From *Simple Wireless* module.

helper/sw-phy-basic-helper.h, .cc – Helper class for `SwPhy` objects. From *Simple Wireless* module.

sw/model/:

model/kmrf2-address.h, .cc – Definition of addresses for KMRF2.

model/kmrf2-layer-headers.h, .cc – Definition of the header and payload structures.

model/kmrf2-app.h, .cc – Definition of the KMRF2 meter and concentrator applications.

model/kmrf2-network.h, .cc – Definition of the KMRF2 Network Layer.

model/kmrf2-data-link.h, .cc – Definition of the KMRF2 Data Link Layer.

model/kmrf2-mac.h, .cc – Definition of the KMRF2 MAC Layer.

model/kmrf2-lists.h, .cc – Definition of various lists, such as local list and routing table.

model/kmrf2-net-device.h, .cc – Definition of `Kmrf2NetDevice`. Based on *Simple Wireless* module.

model/mbus-app.h, .cc – Definition of WM-Bus meter and concentrator applications.

model/sw-channel.h, .cc – Definition of `SwChannel`. Based on *Simple Wireless* module.

model/sw-mac-csma.h, .cc – Definition of `SwMacCsma`. Based on *Simple Wireless* module.

model/sw-mac.h – Definition of `SwMac`. Based on *Simple Wireless* module.

model/sw-phy.h, .cc – Definition of `SwPhy`. Based on *Simple Wireless* module.

model/sw-net-device.h, .cc – Definition of `SwNetDevice`. Based on *Simple Wireless* module.

model/node-loading-functions.h, .cc – Functions for loading nodes from the KML files with coordinates of the meters and concentrators.

model/geography-functions.h, .cc – Functions for converting the geographic coordinates into Cartesian coordinates.

model/ConcentratorMappingsUtils.h, .cc – Functions for loading mappings between concentrator names, serial numbes and IP addresses.

model/kmrf2-tags.h, .cc

model/sw-mac-header.h, .cc

other – `angles.h`, `.cc`, `antenna-model`, `parabolic-antenna-model`, `.cc`, `isotropic-antenna-model`, `.cc` were copied from development version of NS-3.14, but are mostly not used.

sw/test/:

test/kmrf2-test-suite.cc – Suite of several tests for the KMRF2 model.

sw/doc/:

doc/sw.rst – Document from the original package of the *Simple Wireless* module.

E.3 Utility Programs

Other programs were created in order to help work with the data and simulations, but are not interacting with NS-3 directly.

Utility/:

analyzelogs.cc – Loads a database with measurements and outputs a CSV summary that can be used elsewhere to the standard output. Superseded by the `mbus-simple-ex` option `-printDistances=1`.

latestTrace2csv – Loads the output database of the WM-Bus simulation and creates a CSV file from the latest traces. The CSV file can be used in `packethit.m`

crawllogs.cc – Gets data from hourly databases into one master database.

parseDB.sh – Shell script that crawls the given directory and uses all the found databases as an input to `crawllogs`.

ConcentratorMappingsUtils.cc – Mappings between concentrator names, IP addresses and serial numbers.

packetHit.m – Displaying the packet hit.

plotacc.m – Script to plot the changing delay of the WM-Bus transmissions.

propagationfitOld2.m, **propagationfit.m** – Scripts for testing propagation model and fitting the data.

USAGE – Instructions for preparing the measurement databases.

E.4 Sample Data

Data/:

concentratornames.csv – Short and long names of the concentrators.

concentrator-serial-ip.csv – Serial numbers and IP addresses of the concentrators.

concentratorsgeo.kml – Positions of the concentrators.

metersgeo.kml – Positions of the meters.

outputdb_means.sqlite – Measured path losses and hit rates, saved in SQLite.

outputdb_means.sql – Measured path losses and hit rates, exported to SQL.