oTWLAN – a tool to simulate tactical ad-hoc networks

Tore J. Berg

Forsvarets forskningsinstitutt/Norwegian Defence Research Establishment (FFI)

May 15th, 2009

FFI-rapport 2009/00911

1070

P: ISBN 978-82-464-1596-3
E: ISBN 978-82-464-1597-0

Keywords

Modellering og simulering

Datanett

Radionett

Approved by

Eli Winjum	Project Manager
Vidar S. Andersen	Director

English summary

oTWLAN is a stochastic discrete event continuous time simulator developed to assist simulation based research studies of wireless ad-hoc networks. The simulator is primarily designed for analysing link layer and network layer protocol functions. However, a complex radio model is implemented to have an accurate packet capture model. *oTWLAN* has many similar characteristics as WLAN (IEEE 802.11), but also provides a 100kbps radio channel for enlarged radio coverage area and relaying of traffic for increased service coverage area. The network supports multi-level precedence and preemption, which is an important service in military networks and emergency networks.

The first part of this document describes the networks protocols implemented and the design of the simulator. The last part focuses on validation of the simulator and presents some multihop simulation experiments.

Sammendrag

oTWLAN er en nettverksimulator som modellerer et trådløst ad-hoc datanett. Simulatoren er utviklet for å studere nettprotokoller der prioritetshåndtering av brukertrafikk er et krav.

Dette dokumentet viser eksempler på hvordan simulatoren kan brukes, men beskriver også design og realisering av simulatoren.

Contents

1 Introduction

oTWLAN[1] is a stochastic discrete event continuous time simulator modelling a wireless tactical multihop network. The goal of the *oTWLAN* open source project is to develop a simulator suitable for the study of network protocol functions under various network topologies. The simulator is named "tactical" since the network services provided support priority handling of user traffic, which often is a mandatory feature in military and emergency networks. The software supports the entire "life-cycle" of modelling and simulation – functions to implement models of real objects, functions to configure a network, functions to conduct debugging and functions to produce simulation reports.

The *oTWLAN* software is developed under the FFI-project[2] Fundamental Technologies and Trends in Information Security (GOSIKT). An objective of GOSIKT is to study security technologies for system architectures with different bandwidth, battery, processing and memory capacities. One task for this project is to study the transmission capacity required to serve a NATO Public Key Infrastructure (PKI) [25,26] within the tactical domain where only radio based communication is available. The first step of this task is to implement a simulator which models a wireless ad-hoc network supporting physical layer transmission rates from 100kbps to 10Mbps. An ongoing project activity extends the *oTWLAN* simulator with PKI application layer protocols.

The primary target group for the simulator is researchers and students that want to study network protocol functions. This document is written for readers having experience with modelling and simulation of wireless ad-hoc networks. We assume the readers have good knowledge about radio aspects as well as protocol aspects. The modular design of the simulator makes it easy to add new protocol functions in any of the layers as well as modifying the radio characteristics, or add a new radio version. During the design and implementation, we made an effort to assign module and class names that clearly identify their belonging and functionality to ease programming for new users.

oTWLAN versus other simulators. Commercial simulators protect their source code and to modify existing protocols or adding new ones often become difficult. Our research is targeted towards new communication protocols, and simulations of existing protocols are less interesting. Our practical experience is that we need the source code to add trace statements within the code to get insight into network behaviour, and to gather network statistics. *oTWLAN* is based on the open source projects *OMNeT++* [1] and *Qt4* [7]. *OMNeT++* implements the basic services needed for modelling network protocols and build discrete event simulators. *Qt4* is used to build a Graphical User Interface (GUI). *Qt4* is a comprehensive C++ framework for developing cross-platform GUI applications. However, Qt4 has many high quality software modules of general interest. In particular the *Qt4* XML module has been extensively used. Doing simulation experiments are a rather time-consuming task since the user must parameterise a model, interpret the simulation results and produce a report based on many plots from the simulation results. *SimProcTC* [22] is

[1] OMNeT++ based Tactical Wireless Local Area Network
[2] Norwegian Defence Research Establishment, www.ffi.no.

an open source *OMNeT++* based project which implements a tool-chain intended to ease the process of running experiments. *oTWLAN* provides similar functionality where the user works via a set of GUI based editors. The editors produce ".ini"-files and XML-files. These files are the input data to the *OMNeT++* modules.

oTWLAN as a tool. Chapter 2 demonstrates the capabilities of the *oTWLAN* by going through a simulation experiment. *oTWLAN* provides a GUI through which the user can set up the simulator's input data to model a particular scenario. Eager readers, who want to get started with *oTWLAN* simulations as fast as possible, should read chapter 2 first. However, to have an understanding of the network modelled and to be able to configure the network correctly, chapter 3 and 5 have to be read. The simulator is designed for analysing performance of network protocol functions. It is not designed to conduct radio performance analysis (physical layer protocols), nor to estimate the performance in a specific real terrain.

The radio planning process. An operating scenario is modelled by specifying radio link pathloss models, transmitting power, etc. The user inserts a number of nodes on the playground and then sets a number of radio parameters to get the radio coverage wanted. *oTWLAN* provides a number of interactive management tools by which the user can predict Signal-Noise-Ratio (SNR) values and the resulting success probabilities of the different parts of the radio frame.

The network planning process. The *oTWLAN* MAC protocol is a random access protocol, which means that packet collisions may occur. The collision probability increases with increasing traffic and therefore the network layer experiences change in topology; links having good quality may become useless as the traffic increases. *oTWLAN* has implemented a set of graphical tools facilitating deployment of network nodes on the playground, setting up routing matrix and inserting traffic generators above OSI layer 7.

oTWLAN is a hierarchical model. oTWLAN models a complex multihop network and the system must be split into functional units of smaller size suitable for implementation in a programming language. To achieve this, the system is broken down into smaller components in a structured manner. *oTWLAN* is at the top level divided into a horizontal layer model in conformance with the OSI Reference Model (chapter 3). We specify the layer protocols and queue structures for the network nodes. The next step is to specify the simulator's main data structure that models a network of radio nodes. This is done in chapter 4 "Modelling a Network of Radios" where we end up with leaf nodes in a data tree. These leaf nodes are called "simple modules" in *OMNeT++* terminology [1] and are coded in C++.

oTWLAN as a framework. Users planning to use this project as a framework want to implement new protocols or reuse part of the GUI code in other projects. We have made a great effort to ease this task by assigning C++ class names that easily can be related to their functionality and the software module they belong to. Chapter 8 "The Software Architecture" specifies the naming convention used for software modules, class names and file names. This document is addressed both to users of *oTWLAN* as well as programmers. To ease the navigation within the source code,

we have included class names in the text within this document. Hopefully, this simplifies the task of locating particular functions in the source code. Chapter 7 gives some guidance on how to remove certain software components. An automatic documentation system (*doxygen*, www.doxygen.org) has been applied, so this document gives only an overview of the implementation. The starting point for a detailed description of the implementation is the *doc/html/index.html* file.

2 Learning to Drive

The objective of this chapter is to exemplify the functionality of the *oTWLAN* simulator. We use an example based on a hypothetical scenario where nine nodes are deployed on the playground shown in Figure 2.1.

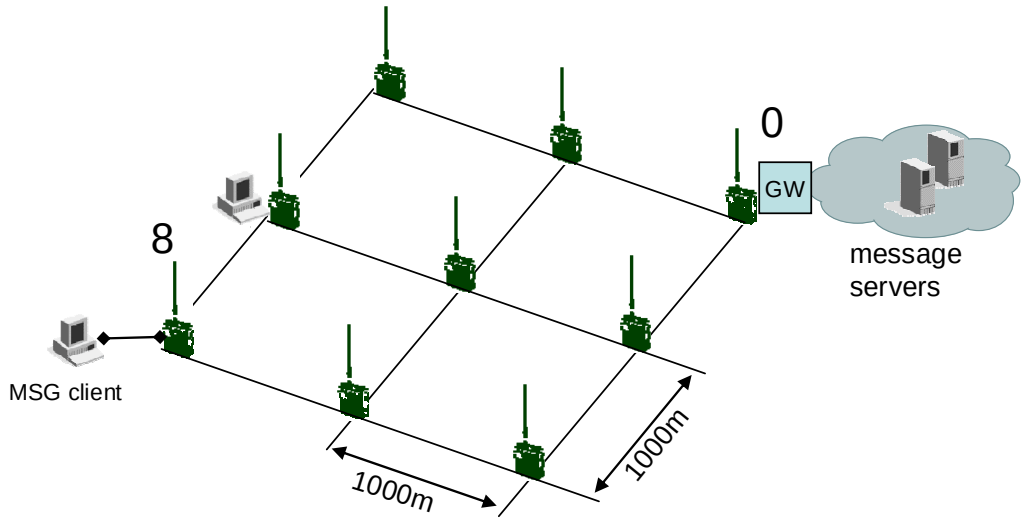


Figure 2.1 The playground and nine nodes placed on a regular grid of size 2000m x 2000m.

One terminal is attached to each radio node and all the terminals run the same application; a message client that sends short messages to the other clients in the network. All the sites are identical with the exception of node 0, which operates as a gateway to message servers. If the users shall communicate with users outside the wireless zone, the traffic must pass through node 0. We assume the message system is based on User Datagram Protocol (UDP) services.

Our task is to assist the users with the network planning constraint by a set of user requirements. The users want to keep the transmitting power at minimum to maximize battery lifetime and to have a low visibility in the RF spectrum. However, they are anxious to be able to access the message servers for external communications. We prescribe activation of the ARQ protocol on all radio links to increase the resilience against packet loss. The link between node 0 and node 8 is the longest link in this network, and we must carefully consider this link during the radio planning process (section 2.2).

The users specify that their terminals (OSI layer 7) generate message traffic according to Table 2.1. The outcome of our analysis shall be a recommended power level and throughput/delay plots as function of the offered message traffic.

Parameter	Value
Message arrival distribution	exponential, variable mean
Payload distribution	fixed 400 bytes
Pattern	uniformly distributed
Priority distribution {P0(lowest),P1,P2,P3}	{0.4, 0.3, 0.2, 0.1}

Table 2.1 Traffic data for user terminals (offered traffic to layer 7).

Traffic pattern "uniformly distributed" means that the nodes address any other of the nodes with the same probability. *oTWLAN* supports Multi-Level Precedence and Preemption (MLPP) [17,18] and the priority distribution tells the relative volume of each level. For example, the lowest priority level P0 amounts to 40% of the total traffic volume. Some radio parameters are also needed and this example takes the values from Table 2.2.

Parameter	Value
Radio hardware	1Mbps version
Antenna height	2m
Antenna gain (tx/rx)	0 dBi
Pathloss model	Egli

Table 2.2 Radio parameters. oTWLAN supports the three radio versions specified in section 3.4.

The following sections describe the steps required to set up the simulator's input data. It is important to execute the steps in the same order as they are presented since the steps depend on each other. For example, you cannot create the traffic generators in section 2.4 for nodes that do not exist on the playground in section 2.1.

2.1 Creating the Playground

A playground is the xy-plane where the network nodes are deployed. The "New Simulation Scene Editor" in Figure 2.2 provides functions to set up a playground and insert a number of nodes. We set the playground size to 5000m x 5000m, specify 9 nodes and click on the "Create"-button.

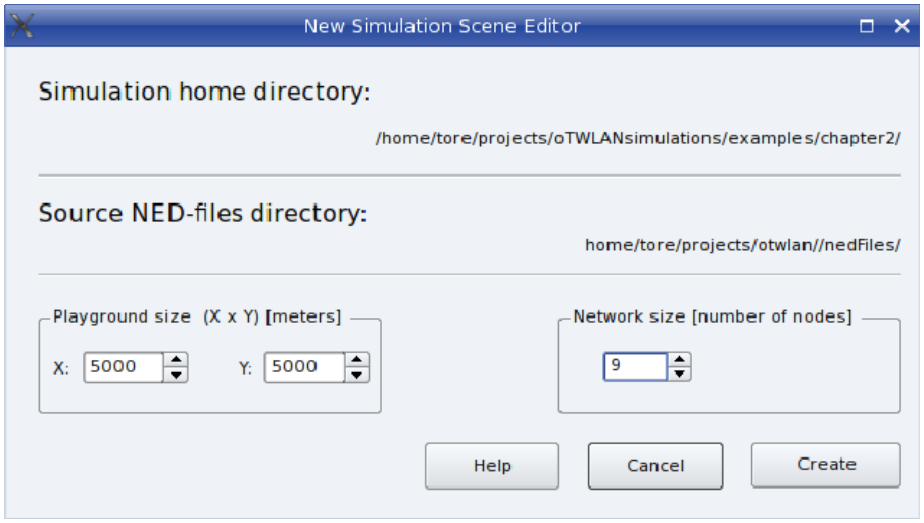


Figure 2.2 The menu "Project->New" invokes an editor for setting up a playground (class GUI_NewSceneEditor).

Then the *oTWLAN* dashboard pops up and visualises the nodes as red rectangles, see Figure 2.3. A user of *oTWLAN* may have a mental model which describes the task of deploying wireless networks and often this model arises from real-world experience. The dashboard design attempts to reflect the user's mental model by setting focus on the physical equipment by displaying the playground area in the main widget. The user sees the geographical locations of the radios deployed and he may select one or more of them and drag them to new positions.

In our scenario, the nodes shall be placed on a grid and this can be achieved through the drag-and-drop functionality, or by writing the nodes' positions into a table. By selecting a node and then right-clicking on the mouse button, the table widget pops up. Remember to click the "Update Data" button in the upper right corner of the dashboard when the playground is ready. The underlying data model is not changed before this button is clicked. Then you can use the menu "Project->Save" to save your playground to a file (*setup/playground.xml*). Figure 2.3 shows the playground after the data has been set correctly. An existing project can be opened by the menu "Project->Open".

The next step is to specify the pathloss model by means of the pathloss editor (menu "Editors->Pathloss"). This is easy in our example since all the links shall use the Egli pathloss model [9]. *Figure 2.4* presents the picture after correct setting of the pathloss data. The pathloss matrix pops up after one click on the "Show Matrix" button. Click on the "Save"-button and the pathloss matrix is written to a file (*setup/pathloss.xml*).

Now nine nodes exist on the playground and we have created a "terrain" by specifying a pathloss model. The next step is to configure the radio attributes to have the radio coverage area wanted.

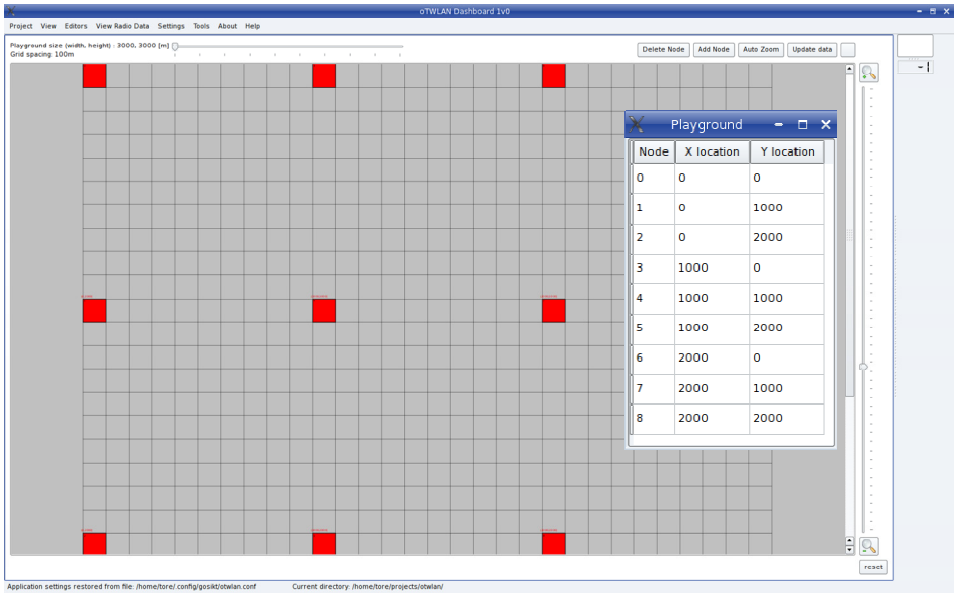


Figure 2.3 The oTWLAN dashboard (class GUI_MainWindow) provides a set of functions to set up simulation scenarios, run simulations and analyse results.

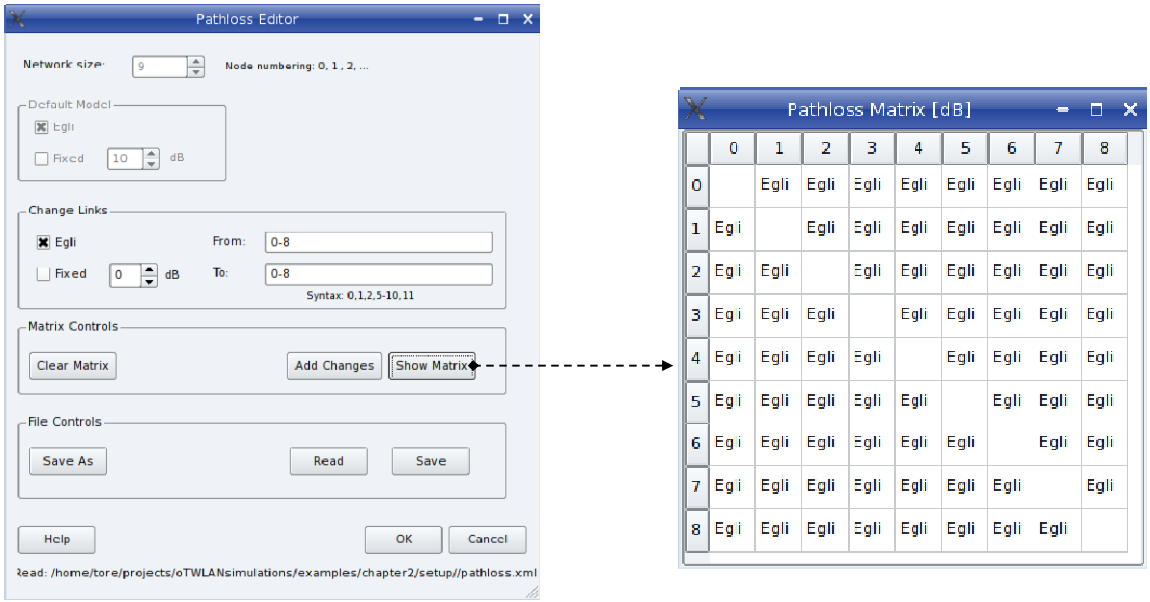


Figure 2.4 The pathloss editor (class GUI_PathlossEditor) facilitates setting of a pathloss model on a per link basis.

2.2 Radio Planning

The physical layer editor facilitates selection of different radio hardware and configuration of transmitter and antenna parameters. The first step is to select the radio hardware and we shall select the 1Mbps option in the editor's "Global Parameter" page, see Figure 2.5 (left picture). Then the transmitter and antenna parameters are set from the "Default Parameters"-page (right

picture). All the radios in our scene shall have the same settings, so this example does not need to open the ”Node Parameters”-page.

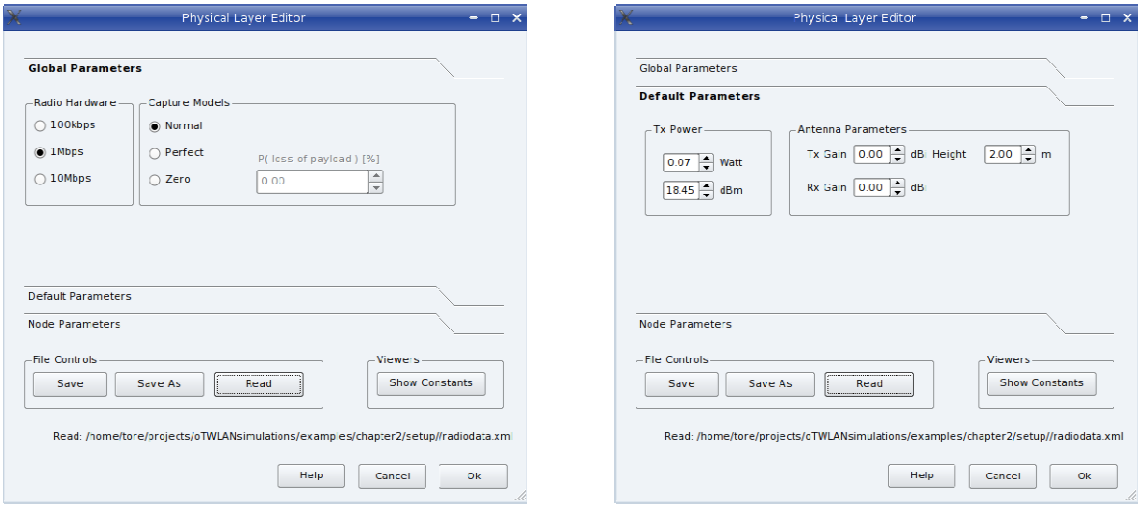


Figure 2.5 The physical layer editor (class GUI_PhyLayerEditor) has three tool box pages which provide selection of radio hardware and specification of transmitter and antenna parameters. Individual node parameters can be set from the ”Node Parameters”-page.

Different sections of the radio frame have various ranges since they are carried by a different number of chips (see Table 3.1). The preamble is carried by the largest number of chips and gets the longest link range. A successful packet reception occurs when the radio header and the radio payload are correctly received. It is the Symbol Error Rate (SER) and the maximum packet length that determines if a radio link gets a reasonable retransmission rate. The *oTWLAN* Direct Sequence Spread Spectrum (DSSS) radio sends 8 information bits in a symbol. Figure 2.6 indicates the packet loss probability 0.4 at a distance of 2.5 km when sending a 500 byte packet using 10 dBm (10 mW) transmitting power.

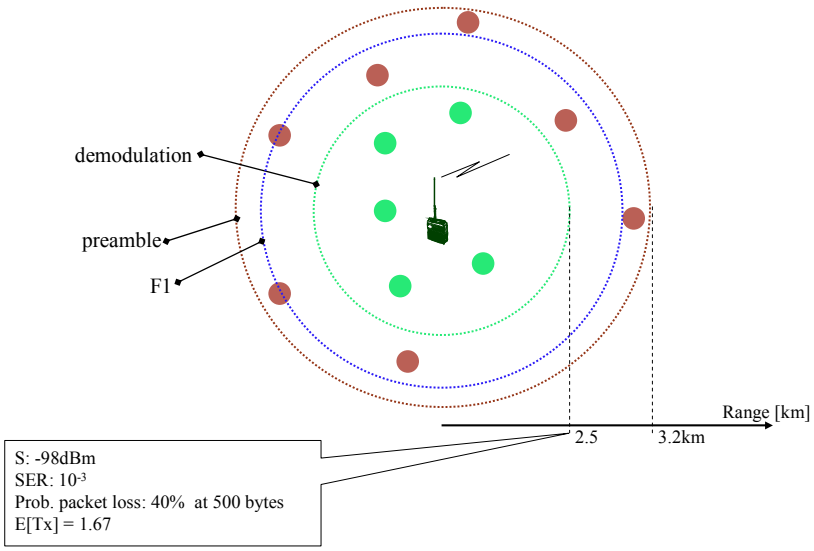


Figure 2.6 Radio frame coverage prediction when sending at 10 dBm (10 mW).

Our first task is to find the minimum transmitting power constrained by the users' requirement of reaching node 0. *oTWLAN* supports relaying and we could set the power level to have a low SER at 1000m and all nodes are connected. However, the *oTWLAN* uses a random access protocol (section 3.3) and the system will suffer from collisions due to the hidden-node problem as the traffic level increases. Hidden nodes and relaying consume much capacity and lead to a significant reduction of the throughput capacity. *oTWLAN* provides a "RF Link Data Viewer" (Figure 2.7) which is useful for inspecting the RF link quality. *Figure 2.8* presents the situation when the transmitting power is set to 10 mW. The lower left table expresses the probability to detect the preamble correctly and shows that the link $0 \leftrightarrow 8$ fails 35 times out of 100 (the detection probability is 0.65). This means node 0 and node 8 have different understanding of the channel state (busy or idle) a significant part of the time. The network can, of course, cope with this situation but the MAC protocol has lower efficiency since this is a preamble sense protocol[3]. A preamble detection failure implies that a node assumes the channel is idle and acts accordingly. If we look at the (1-SER)-table at the right in *Figure 2.8*, we see that the payload part never succeed. We also note the low probability of success for the other links. Remember that we shall transfer long packets[4].

We want to increase the power level beyond 10mW to achieve a good link quality on the link between node 0 and 8 under low traffic conditions where the background noise is the dominating cause of payload corruption. Then we have built in a margin to the problems that occur as the offered traffic increases as well as improved resistance to network jamming and interference from other transmitters.

We decide to dimension the transmitters such that the average number of transmissions needed for successful delivery is below 2 for all the radio links during the low traffic hour. Then maybe

[3] Like IEEE 802.11. A carrier sense protocol can detect a busy channel at any part of the air frame.
[4] *oTWLAN* supports segmentation and reassembly, and we could reduce the air frame size, but the drawback is more overhead per user message.

we are able to have a reasonable retransmission rate during the high traffic hour. This is verified later in this chapter.

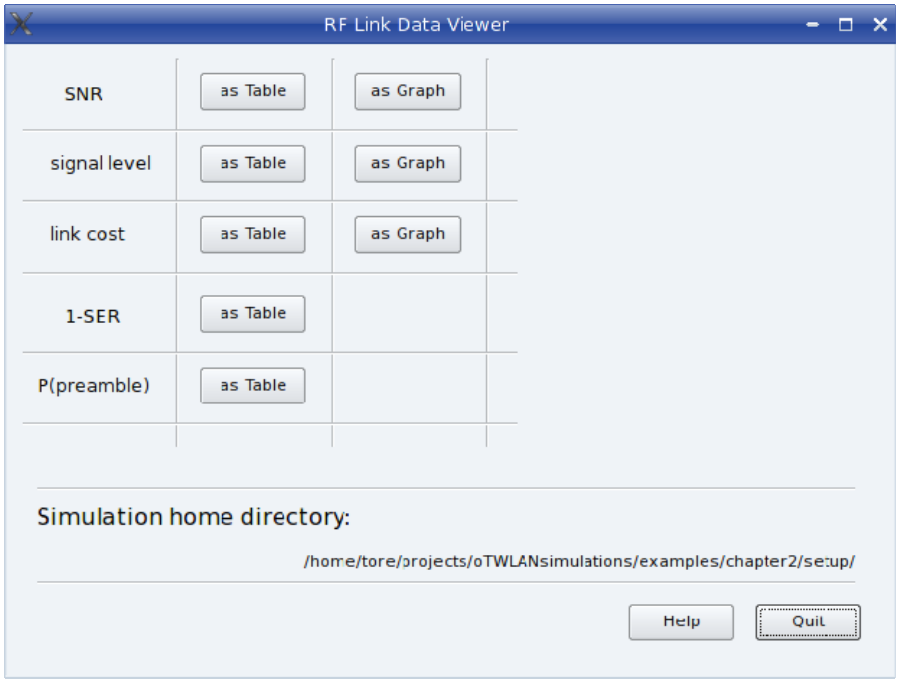
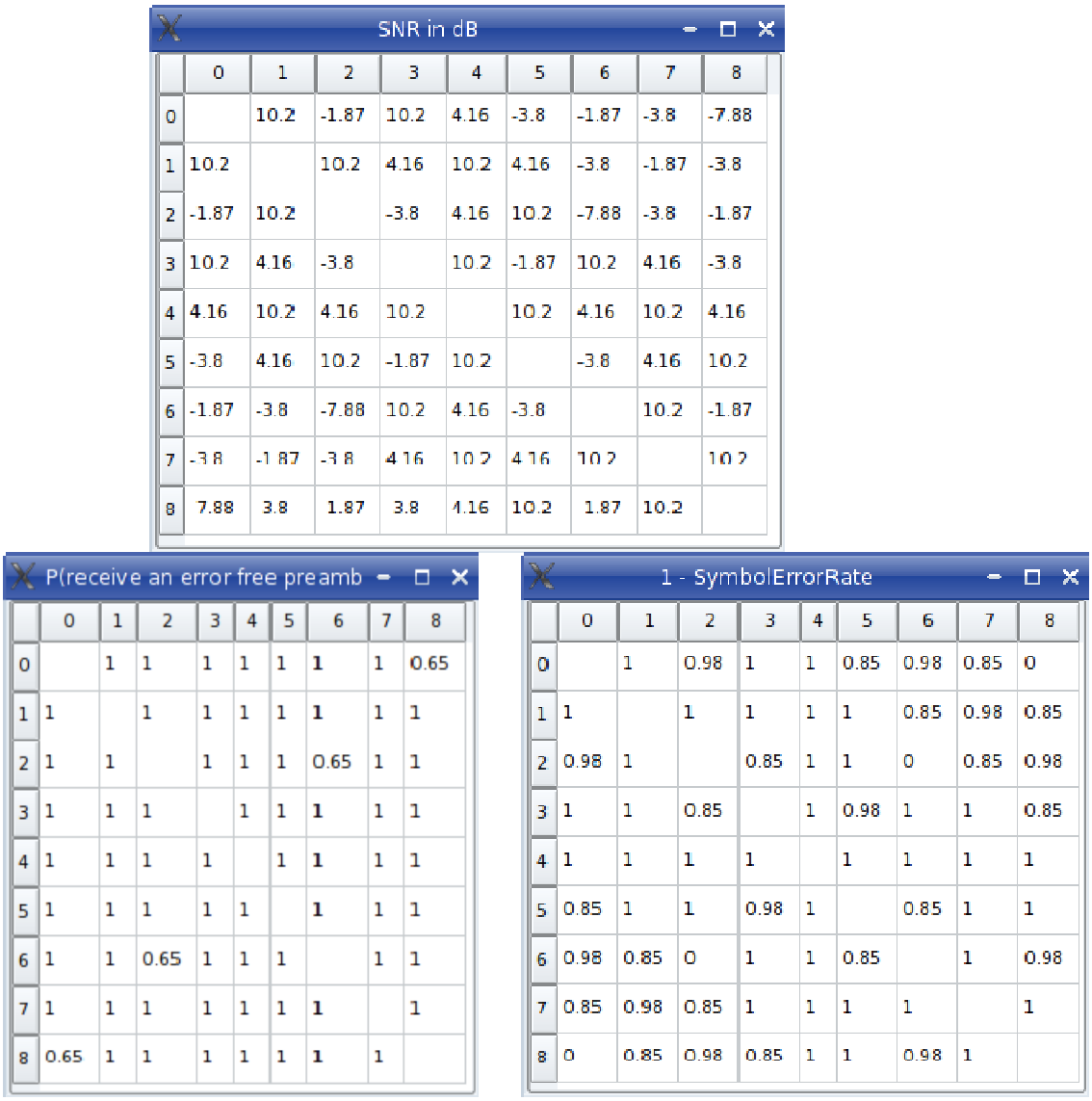


Figure 2.7 The "RF Link Data Viewer" (class GUI_RfLinkViews) provides widgets to assist the user in the radio planning phase. The widget is activated from the menu "View Radio Data"->"Rf Link Data".

The message size is 400 bytes and the *oTWLAN* protocol stack adds 22 bytes Protocol Control Information (PCI) [14] such that the payload size on the air becomes 422 bytes. The probability to deliver the packet in a single transmission is $(1-ser)^{422}$ and this value must be larger than 0.5. Therefore the SER must be lower than 0.0016. From the demodulation table for the radio (class *RadioSER*), we find that the demodulation SNR limit is somewhere between 0 dB and +1 dB. We must then analyse the link budget on the longest network link (2828m) and set the transmitting power to fulfil this SNR limit. A link budget analysis shows that 63 mW gives a radio link of sufficient quality and we prescribe use of 70 mW. Note that the boundaries shown in Figure 2.6 are represented by probability distributions[5] in the simulator.

Figure 2.9 presents the situation at 70 mW and shows an increase in Signal-to-Noise Ratio (SNR) on the link $0 \leftrightarrow 8$ from -7.8 dB to 0.6 dB, the preamble can be detected reliably in the entire network and the SER is low. The table does only use 3 digits and a better way to inspect the link quality is through a link cost widget. This is a subject for section 5.4 that deals with routing.

[5] For example, the preamble has 4.5 dB dynamic range. SNR levels below -10.5 dB are never detected while SNR levels above -6.0 dB is always detected (class *RadioSER*).



SNR in dB

	0	1	2	3	4	5	6	7	8
0		10.2	-1.87	10.2	4.16	-3.8	-1.87	-3.8	-7.88
1	10.2		10.2	4.16	10.2	4.16	-3.8	-1.87	-3.8
2	-1.87	10.2		-3.8	4.16	10.2	-7.88	-3.8	-1.87
3	10.2	4.16	-3.8		10.2	-1.87	10.2	4.16	-3.8
4	4.16	10.2	4.16	10.2		10.2	4.16	10.2	4.16
5	-3.8	4.16	10.2	-1.87	10.2		-3.8	4.16	10.2
6	-1.87	-3.8	-7.88	10.2	4.16	-3.8		10.2	-1.87
7	-3.8	-1.87	-3.8	4.16	10.2	4.16	10.2		10.2
8	7.88	3.8	1.87	3.8	4.16	10.2	1.87	10.2	

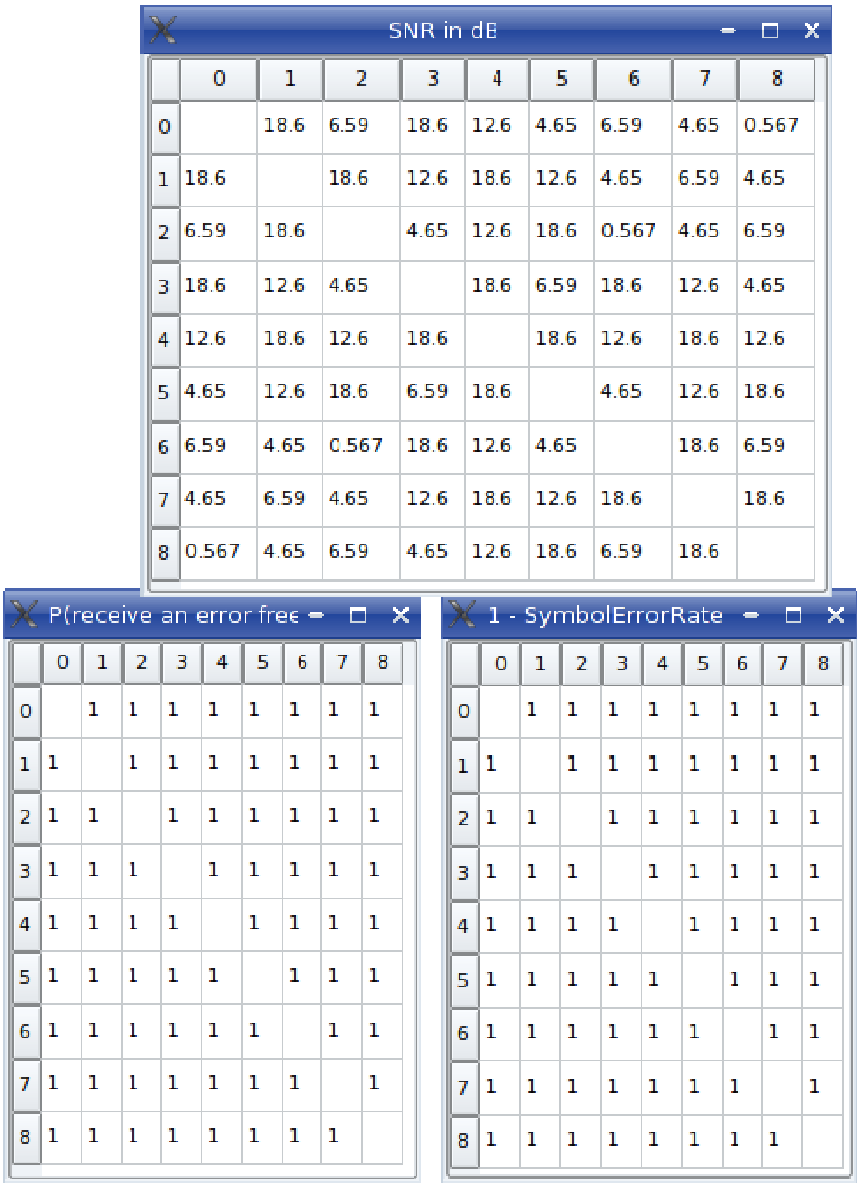
P(receive an error free preamb

	0	1	2	3	4	5	6	7	8
0		1	1	1	1	1	1	1	0.65
1	1		1	1	1	1	1	1	1
2	1	1		1	1	1	0.65	1	1
3	1	1	1		1	1	1	1	1
4	1	1	1	1		1	1	1	1
5	1	1	1	1	1		1	1	1
6	1	1	0.65	1	1	1		1	1
7	1	1	1	1	1	1	1		1
8	0.65	1	1	1	1	1	1	1	

1 - SymbolErrorRate

	0	1	2	3	4	5	6	7	8
0		1	0.98	1	1	0.85	0.98	0.85	0
1	1		1	1	1	1	0.85	0.98	0.85
2	0.98	1		0.85	1	1	0	0.85	0.98
3	1	1	0.85		1	0.98	1	1	0.85
4	1	1	1	1		1	1	1	1
5	0.85	1	1	0.98	1		0.85	1	1
6	0.98	0.85	0	1	1	0.85		1	0.98
7	0.85	0.98	0.85	1	1	1	1		1
8	0	0.85	0.98	0.85	1	1	0.98	1	

Figure 2.8 Radio coverage predictions at transmitting power 10 dBm (10mW).



	0	1	2	3	4	5	6	7	8
0		18.6	6.59	18.6	12.6	4.65	6.59	4.65	0.567
1	18.6		18.6	12.6	18.6	12.6	4.65	6.59	4.65
2	6.59	18.6		4.65	12.6	18.6	0.567	4.65	6.59
3	18.6	12.6	4.65		18.6	6.59	18.6	12.6	4.65
4	12.6	18.6	12.6	18.6		18.6	12.6	18.6	12.6
5	4.65	12.6	18.6	6.59	18.6		4.65	12.6	18.6
6	6.59	4.65	0.567	18.6	12.6	4.65		18.6	6.59
7	4.65	6.59	4.65	12.6	18.6	12.6	18.6		18.6
8	0.567	4.65	6.59	4.65	12.6	18.6	6.59	18.6	

	0	1	2	3	4	5	6	7	8
0		1	1	1	1	1	1	1	1
1	1		1	1	1	1	1	1	1
2	1	1		1	1	1	1	1	1
3	1	1	1		1	1	1	1	1
4	1	1	1	1		1	1	1	1
5	1	1	1	1	1		1	1	1
6	1	1	1	1	1	1		1	1
7	1	1	1	1	1	1	1		1
8	1	1	1	1	1	1	1	1	

	0	1	2	3	4	5	6	7	8
0		1	1	1	1	1	1	1	1
1	1		1	1	1	1	1	1	1
2	1	1		1	1	1	1	1	1
3	1	1	1		1	1	1	1	1
4	1	1	1	1		1	1	1	1
5	1	1	1	1	1		1	1	1
6	1	1	1	1	1	1		1	1
7	1	1	1	1	1	1	1		1
8	1	1	1	1	1	1	1	1	

Figure 2.9 Radio coverage predictions at transmitting power 18.4 dBm (70 mW).

2.3 Creating the Routing Table

oTWLAN does not implement a routing protocol but must have a routing table to find paths between the network nodes. The routing table for the current scenario is a very simple table since all the end-destinations are reached in one hop. However, the user must still create a routing table, and the “Routing Viewer” in Figure 2.10 assists the user with this task. Simply press the “Build Minimum Spanning Tree”-button, answer yes on the question popping up and the routing table is saved to a file (*setup/routing.xml*). Section 5.4 gives detailed information about routing.

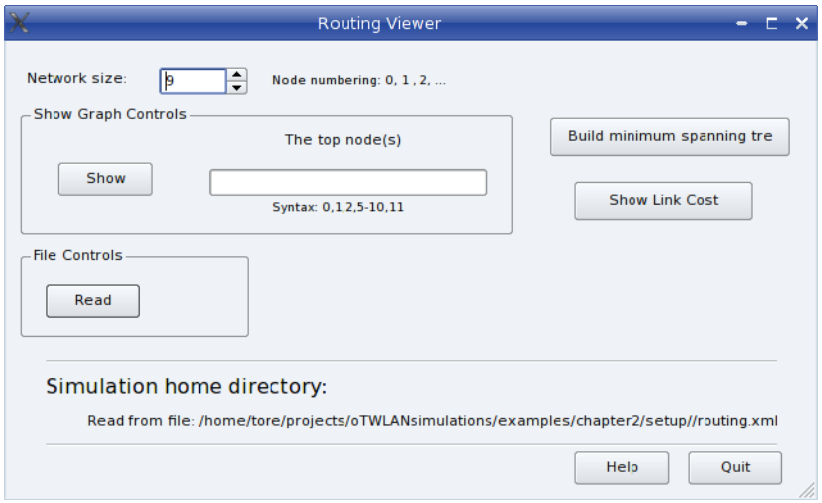


Figure 2.10 The "Routing Viewer"-widget (class GUI_Routing).

2.4 Creating the Traffic Generators

A typical simulation experiment for us is to estimate the expectations of stochastic variables as some input parameters are varied over a set of values. This is exemplified by Figure 2.11 where the estimate of interest is the average end-to-end delay as function of the offered traffic taken from the set $\{\Lambda_1, \Lambda_2, \Lambda_3, \Lambda_4, \Lambda_5\}$. The index *i* of these elements is referred to as *session run i.* They are all independent of each other, which means that the simulator is set to its initial state at the very beginning of each run[6]. The offered traffic Λ_i will at run-time lead to creation of one or more traffic generators. Normally, the offered traffic set is ordered after increasing load since this eases the process of producing nice output graphs.

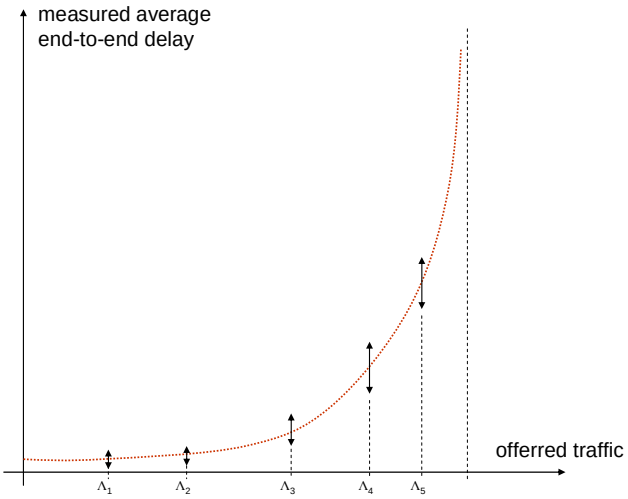


Figure 2.11 A simulation session where the objective is to measure the end-to-end delay as function of the offered traffic. We want to have control of the quality of the estimates and present the measurements as confidence intervals.

[6] All queues are set to an empty state at time instance zero.

The offered traffic is specified in Table 2.1 and our variable is the message arrival rate. To find the levels to use is not a trivial task, especially when the end-to-end delay shall be measured. A practical way to do this is first to find values which give a nice throughput plot and then focus on the delay measurements thereafter. After some trials we found that the following arrival rates will give a nice throughput plot {0.025, 100, 137.5, 150, 162.5, 175, 200, 250, 275, 300, 325, 350, 375, 400, 425, 450, 475, 500, 525} [packets/s]. These 19 session runs are created by means of the "Traffic Editor"-widget in Figure 2.12.

The "P(Arq)" is set to 1 such that all messages enable use of the ARQ protocol function in the network. The lifetime control (section 3.6) is set to 60 seconds since the messages are not outdated after a period of time (for example, a radar application should set the lifetime to 2-5 seconds to prevent relaying of outdated data). The functionality of the traffic editor is too complex to be described in an introductory section and we refer to section 5.3 for a full description.

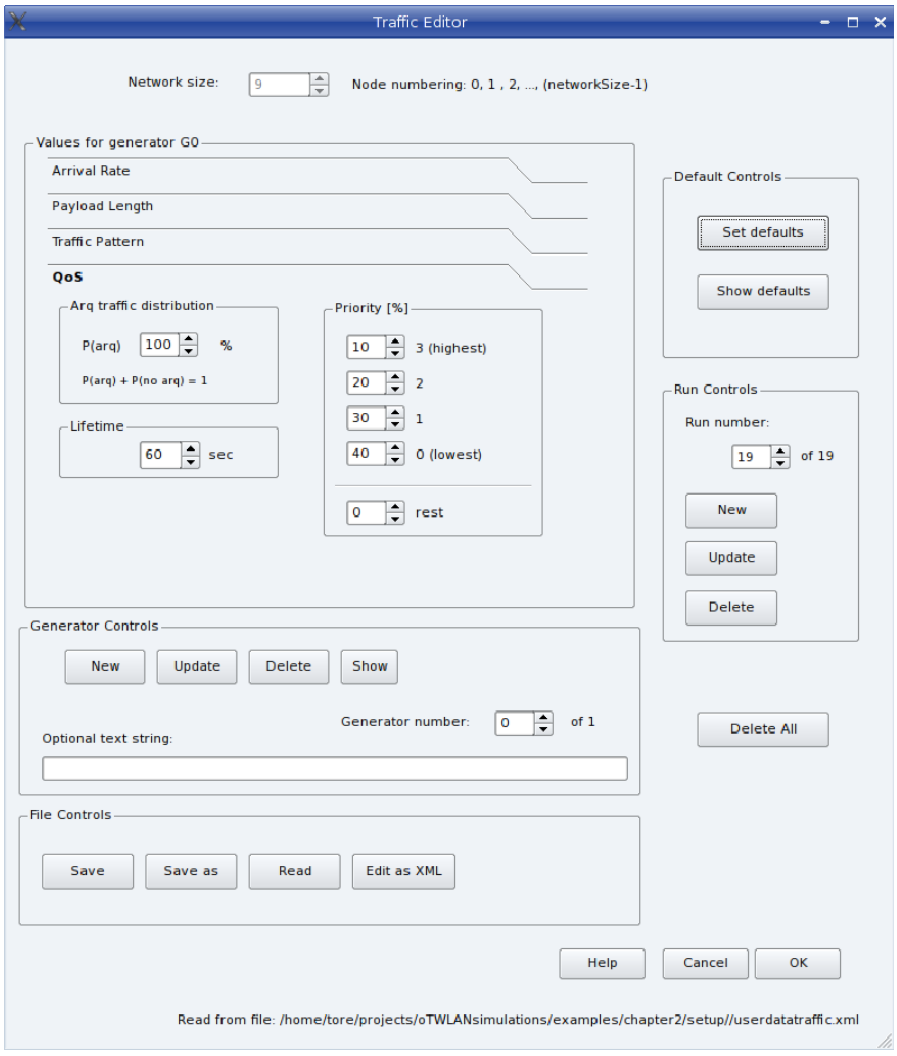


Figure 2.12 The Traffic Editor widget (class GUI_TrafficEditor) is activated by the menu "Editors->Traffic". You must click the Update-button(s) to update the underlying data model after changing an editor attribute.

2.5 Activating Probes

Probes are the objects that take measurements from the stochastic distributions produced by *oTWLAN*. Our project is based on another open source project named *oProbe* [2] for producing statistically sound results. The *oProbe* functionality is provided as an integrated part of *oTWLAN* via the menu "Editors->Probe" (Figure 2.14).

A typical simulation run will have many active probes taking samples from different probability distributions, see Figure 2.13. Unfortunately, network simulations normally imply collecting time-variant and correlated samples[7]. Probes must use controlled statistical sampling techniques to produce trustworthy results, and *oProbe* uses a classical batch means analysis technique [9] to get control of the correlation. When the uncorrelated batch size is found, the probe starts to estimate the confidence interval of the first order moment (the expectation) of the underlying distribution.

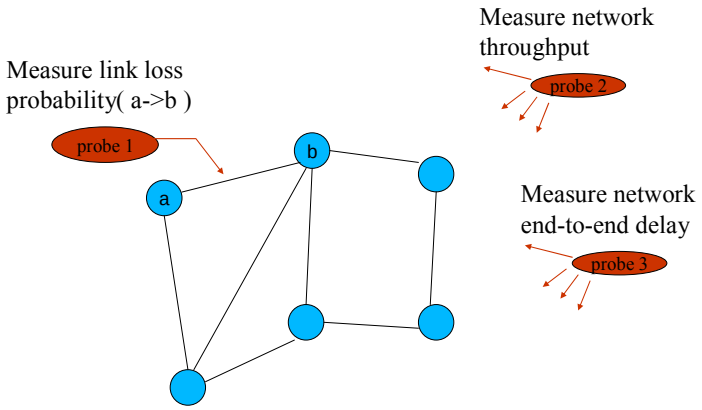


Figure 2.13 Probes are the objects that collect and process samples from probability distributions.

For our scenario, we can immediately conclude that it is impossible to estimate the delay of the lower priority traffic at high traffic loads since the simulation run length explode due to the correlated sample problem[8]. The practical solution is therefore first to set focus on the throughput performance and consider the delay later. Figure 2.14 shows the probe objects used for taking measurements for the throughput plot. Network throughput is the main performance metric of interest, and we set up the probe module to produce a 90% confidence interval. We also measure the throughput per priority levels separately but skip usage confidence intervals. Therefore we must be very careful to use these probes when we analyse the results because we no longer have control of the error in the estimate.

[7] Delay samples from queues are known to have an increasing positive correlation with increasing load level.

[8] Our practical experience is that the variance of the underlying distribution is not the main problem since the simulations terminate immediately when the uncorrelated batch size is reached. We normally use the lag k threshold 0.4; see [2, section 2.2].

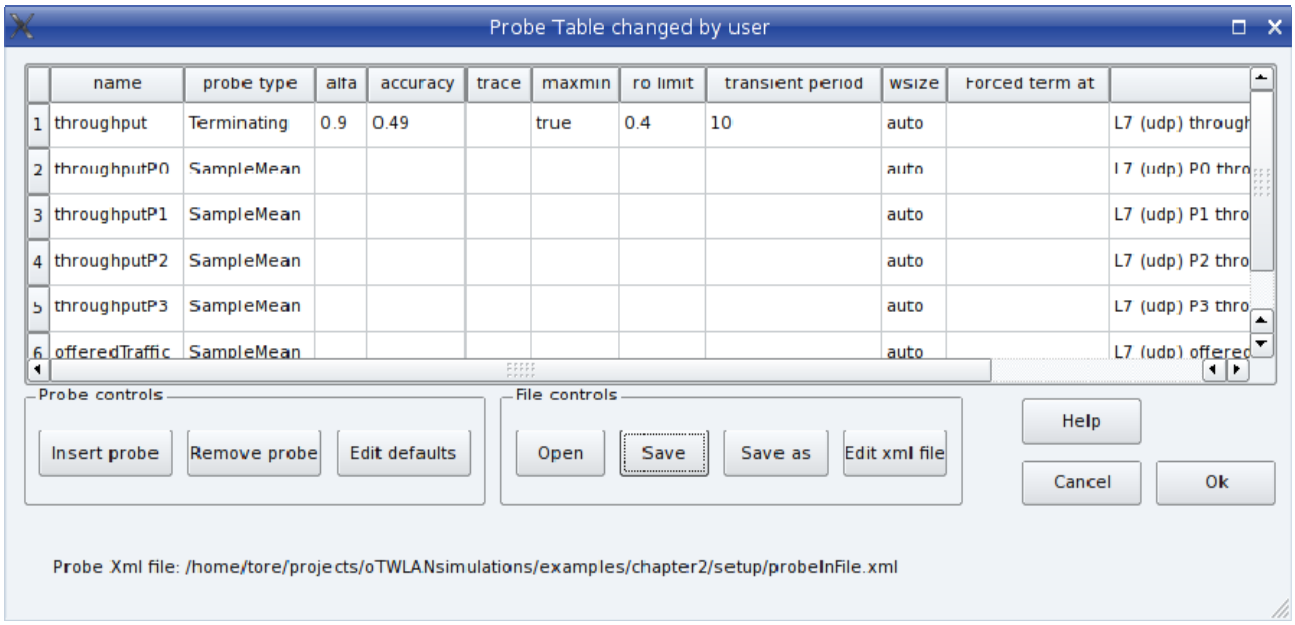


Figure 2.14 The oProbe widget provides functions to activate probes.

2.6 Sanity Checks of Input Data

Most of the simulator's input data is now ready and we have specified 19 session runs, each using a different packet arrival rate. It is irritating to simulate for a long period of time and then crash in run number 18 due to erroneously input data. When the simulator kernel starts, it begins to build the main data structure from the top node (*Sim* in Figure 4.8) and traverses the tree down to the leaf nodes. If an illegal value is found or there exist some inconsistencies, the kernel aborts, usually with a cryptic message for those who do not have detailed knowledge of the software. This section explains how we can circumvent such problems.

The kernel launcher in Figure 2.15 provides functions to set up the run-time environment for debugging or for production, and the following four run-time environments are supported:

Tk-environment
This is a graphical user interface provided by the *OMNeT++* framework to be used for debugging. You get an overview of the network topology and may inspect different parts of the simulator. When you run the simulator, animation of events is activated, and the text window shows a lot of trace messages.

Cmd Debug
This is a text based user interface provided by the *OMNeT++* framework. Only programmers using a symbolic debugger will appreciate the usefulness of this interface.

File Input Check
This is a text based user interface that executes each session run for a short period of CPU time (a few seconds). This will catch many types of input errors such as syntax errors in the XML-based

input files and inconsistent input data (e.g., the playground file declares a network of 10 nodes while the traffic file declares 15 nodes).

Cmd Express

This is the run-time environment to use during the production phase, that is, when you have validated that all the input data is correct and consistent, and that the simulation process will reach a steady-state.

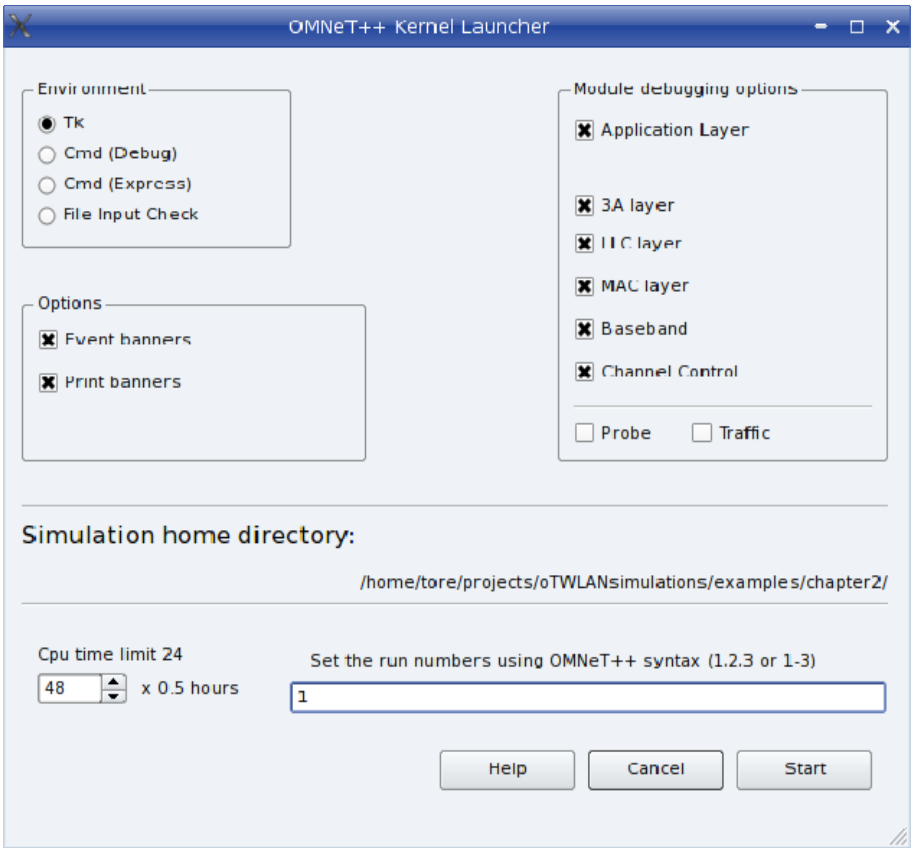


Figure 2.15 The simulator's kernel launcher (class GUI_OmnetStartEditor).

Sanity checks of input data are done in two steps. First select the *Tk*-environment and select the run number with the lowest offered traffic (this is run 1 in our case) and press run on the *OMNeT++* GUI that appears. This step may discover many errors in/between NED-files and XML-files. The next step is to activate the kernel in a "File Input Check"-mode using all the run numbers (this is run 1-19 in our case).

2.7 Running the Experiment

The previous step has checked the input data and at this point we consider them as complete and consistent. Start the simulator in "Cmd-express" mode after you have specified the session run number by inserting the line "1-19" on the run edit line. In this mode the simulator process is optimised for speed and all tracing is turned off.

The *OMNeT++* kernel is started as a separate thread by the main GUI thread. The upper right corner of the dashboard (Figure 2.16) displays a progress bar informing about the status of the thread running. It is impossible to predict the time until termination, so the number does only indicate a healthy thread state and that the probe module collects samples. The digit below the progress bar is the session run number currently executed by the omnet thread (class *oProbe::OMNET_Thread*). The progress bar vanishes when the omnet thread halts.

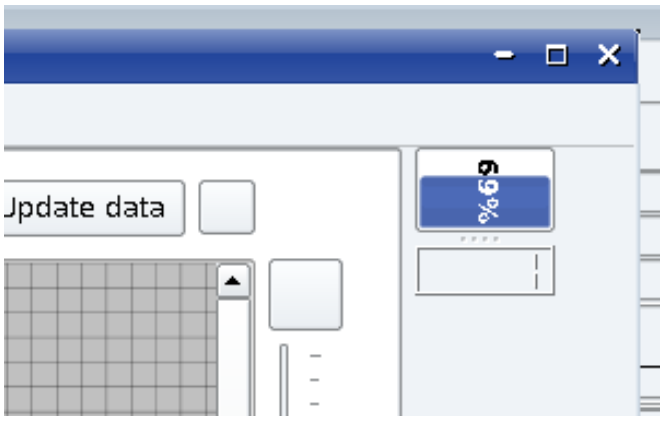


Figure 2.16 The upper right corner of the oTWLAN dashboard displays a simulation progress bar and the ongoing session run number.

2.8 Output Data Analysis

Usually a simulation report contains a plot of data, and the report editor (Figure 2.17) has functions to simplify this work. Reference [2, section 2.1 step 8] explains how the plot in Figure 2.18 can be made by means of the report editor.

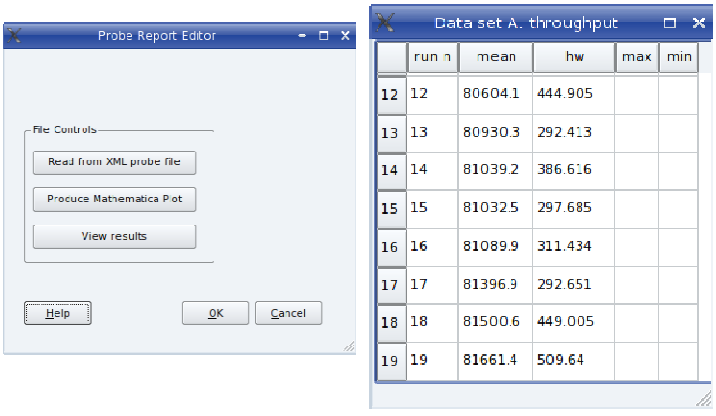


Figure 2.17 The report editor (class oProbe::GUI_ReportEditor).

The *oTWLAN* simulated results are converted to a *Mathematica* [20] compatible data set by means of the *oProbe* report editor. The performance plot in Figure 2.18 is produced by *Mathematica*. The figure shows that the maximum network throughput is 81661 ± 509 bytes/s [9](90% confidence interval) and the network starts to reject the lowest priority traffic when the

[9] 204 messages/s

traffic volume increases beyond 70000 bytes/s (175 messages/s). The throughput plot confirms that the precedence and pre-emption (priority handling) works excellently in this network.

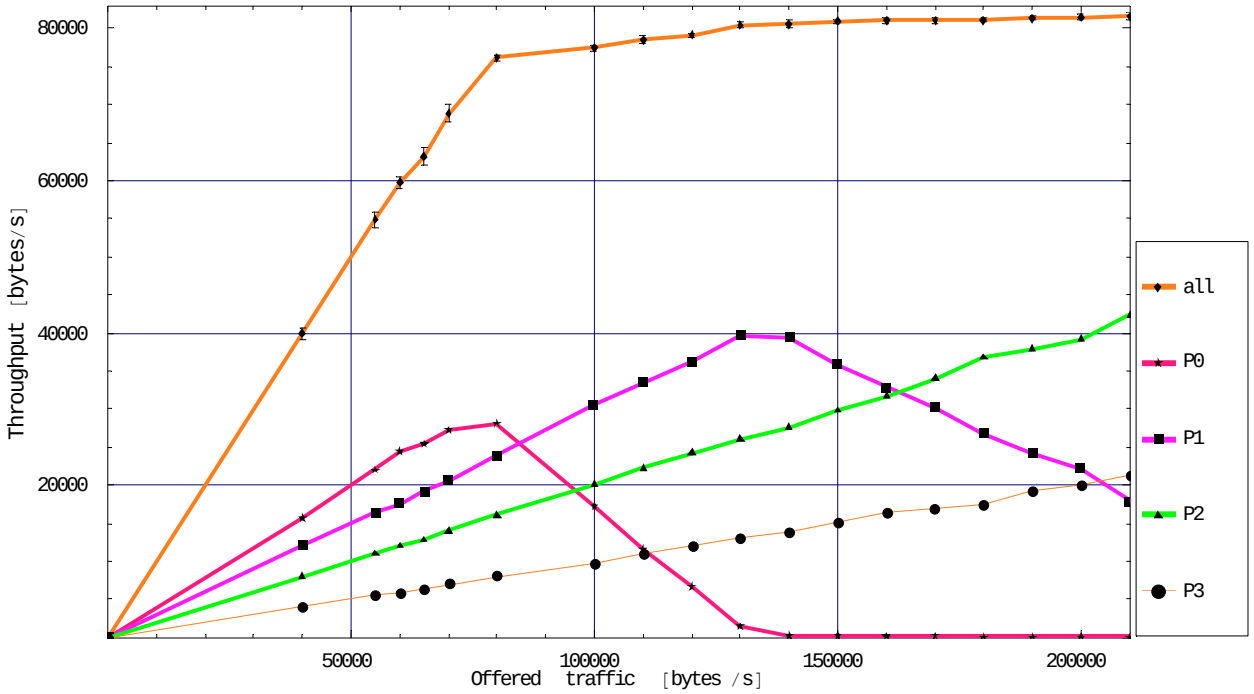


Figure 2.18 Network throughput versus offered traffic. The total throughput is presented as 90% confidence intervals while no confidence control is applied for the other plots (examples/chapter2).

Figure 2.19 shows the delay curve and as expected, the average delay increases fast when the corresponding priority level approaches its capacity limit. Remember that the average delay does not go to infinity since the packets are deleted when their 60 seconds lifetime expire. Three different simulation sessions were needed to conduct this experiment. The first session focused on the lowest priority P0. It excluded the runs above number 6 since the P0 delay probe demands extremely large sample size as the load level increases beyond this run number (offered traffic 70000bytes/s). The uncorrelated batch size[10] at this run number became 600 samples. *oProbe* demands 100 batches of this size and the total sample size at the termination point was 60000 samples (i.e., 60000 messages have passed through the network).

[10] Remember we use the ro limit 0.4 [2].

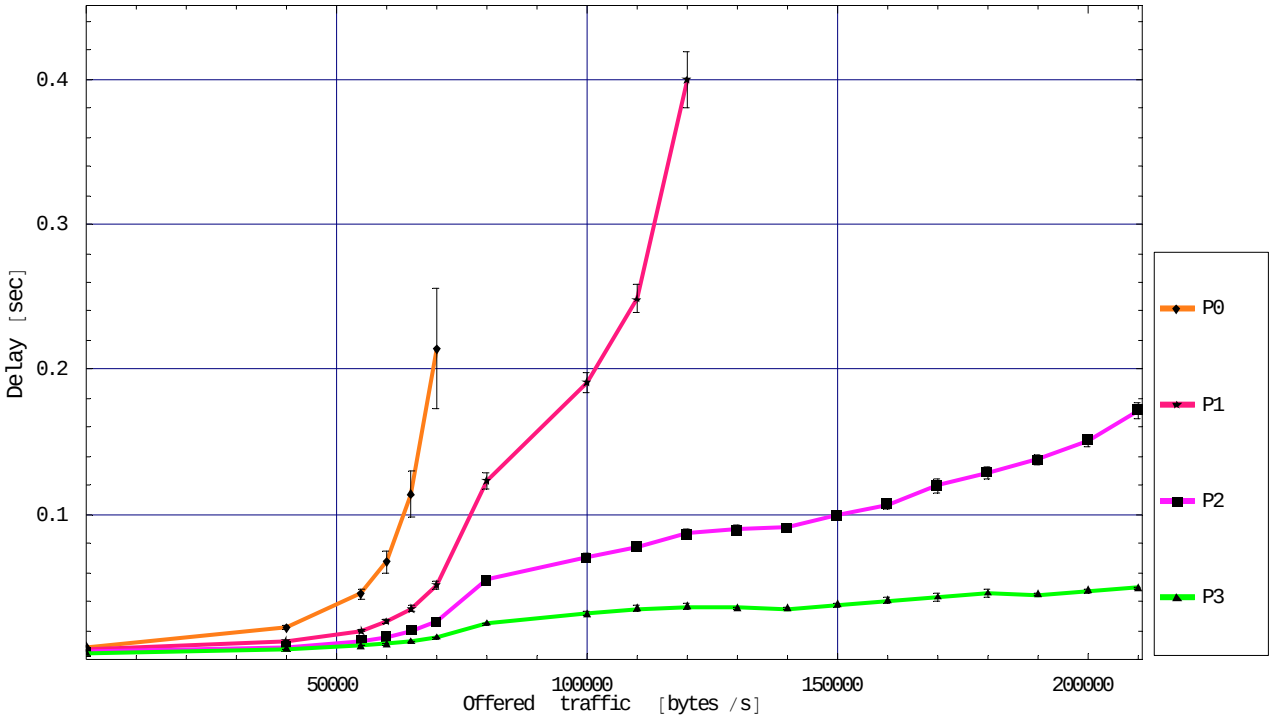


Figure 2.19 90% confidence intervals for the end-to-end delay versus offered traffic.

The second simulation session considered priority level P1 and we altered the P0 delay probe type from "terminating" to "sample mean". All the other delay probes were "terminating" probes. From the throughput plot we conclude that the P1 traffic faces its capacity limit between run number 10 and 11. Thus load levels above run number 10 are excluded from the second simulation session since the simulation run length will be extremely long[11]. The P1 uncorrelated batch size was slightly smaller[12] than for the P0 case.

The third simulation session measured priority P2 and P3 (highest level) simultaneously. As we can see from the throughput plot, neither P2 nor P3 reaches their capacity limits and the simulation session can include the entire run set 1 to 19.

We end this section with some comments about the retransmission rate on the link $0 \leftrightarrow 8$, which was an important subject during the radio planning processes in section 2.2. The transmitting power level was set 70 mW to achieve a packet loss probability below 0.5 on this link during the low traffic hour. The probe which measures the retransmission rate can only sample the network average and we must resort to the counters (section 5.6.3) to get information on a per link basis. If we accept the uncertainty of the statistical accuracy, we can estimate the packet loss probability at node 0 under heavy load as:

$1 - N_{CasAlarms}/N_{rfWavesRxed} = 1 - 53280/58153 = 0.08$

[11] At run number 11 the run length becomes infinite due to the positive correlation problem known from the queuing theory

[12] 550 samples

The counters' output data confirm that it is only the corner nodes {0,2,6,8} that experience packet loss during the low traffic hour, so we expect this value to be a reasonably good estimate despite the lack of confidence control.

3 The Protocol Stack

oTWLAN is designed to model a unicast data service at OSI layer 7 in the terminals shown in Figure 3.1. Traffic generators above layer 7 send packets down to layer 7 (class *L7_DataProtocol*) and depending on the service selected, the traffic is sent over TCP (class *L4_Tcp*) or UDP. The *oTWLAN* protocols are designed to provide the following supplementary subnetwork services:

- Multi-Level Precedence and Preemption (MLPP).
- Lifetime control: the terminals set the maximum lifetime a data packet shall exist in the network[13].
- A service coverage area larger than the radio coverage area: the intra network layer protocols implement relaying.
- Enhanced resilience against packet loss across radio links: the intra network layer protocols implement ARQ.

The MLPP and the lifetime control functions are described in the sections 3.5 and 3.6, respectively. The terminals can enable/disable ARQ on a per packet basis. This is useful since some traffic types (e.g., radar traffic) should not be retransmitted in case of loss while other traffic types benefit from ARQ (e.g., TCP traffic).

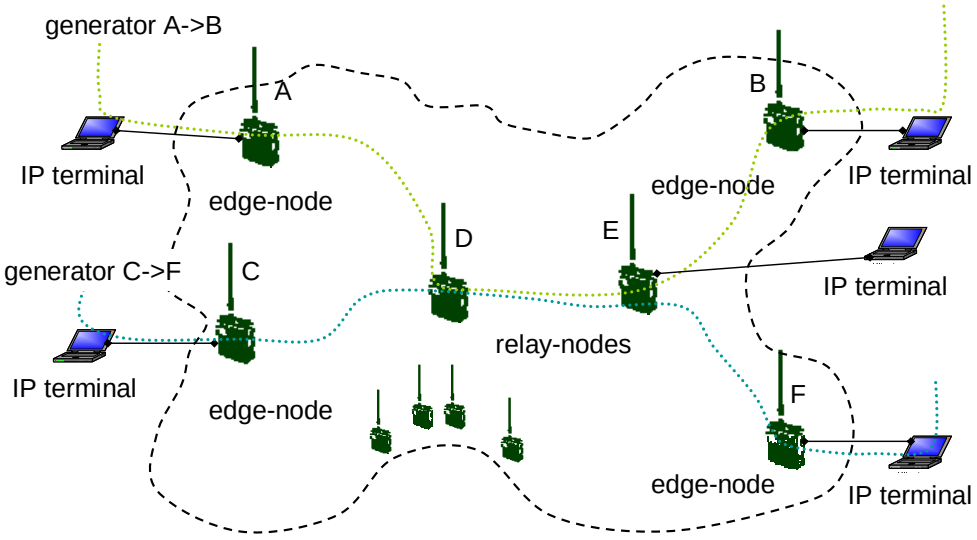


Figure 3.1 An example network with two traffic generators representing the terminal applications.

[13] TCP traffic uses maximum lifetime 60 seconds and enables use of ARQ within the subnetwork. UDP traffic can set these parameters on a per packet basis.

The simulator models the scenario depicted in Figure 3.1 with any number of nodes determined by the input data. An *entry-node* in the figure is a radio node which serves traffic from its local terminal while an *exit-node* is a radio node which delivers traffic to the local terminal equipment. A common name for an entry-node or an exit-node is *edge-node*. A node that relays traffic from an adjacent node is named a *relay-node*. A node may take the role as a relay-node and an edge-node simultaneously. For example, node E in the figure operates as a relay-node and an exit-node.

A generic OSI Reference Model for the network in Figure 3.1 is shown in Figure 3.2 a). A node has two interfaces; a radio based interface and a wire based interface to the terminal equipment. The figure depicts an IP based access protocol between the network and the end-systems. The air interface is not based on IP protocols but these protocols must serve IP traffic between edge nodes. The interface towards the terminal is not modelled since this is a wired based interface with infinite capacity compared to the air interface, and the *oTWLAN* protocol stack is simplified to the stack shown in Figure 3.2 b). The forthcoming sections describe the protocol functions in each layer. The 3b layer protocol is not included in the simulator.

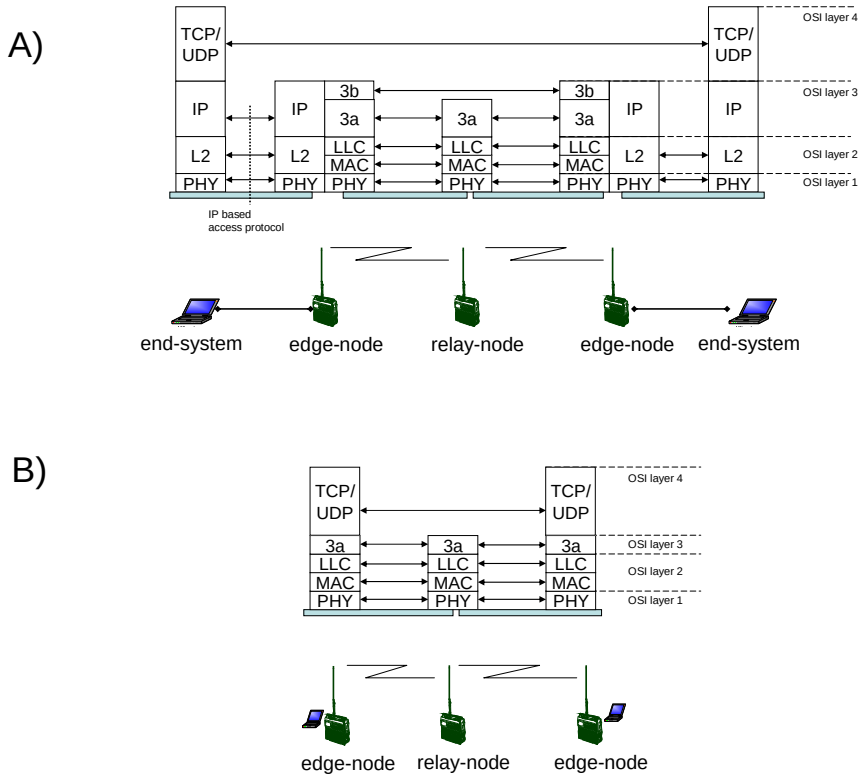


Figure 3.2 Layered models. Figure a) shows the protocol stack for a real system while b) shows the stack implemented in the simulator. The 3b layer protocol carries layer 3 information which is not needed by relay nodes but must be signalled end-to-end.

3.1 The 3a Layer

The 3a layer performs store and forwarding operation in multihop networks and the following functions are implemented:

- Data transmission using ARQ and passive acknowledgement[14] [9]
- Data transmission without ARQ
- Duplicate filtering
- Lifetime control (is described in 3.6)
- Precedence and preemption (is described in 3.5)
- Segmentation and reassembly
- Relaying
- Flow control

The paragraphs below give a short description of these protocol functions.

ARQ and Passive ACK

A "last hop" *PDU* (Protocol Data Unit) is a PDU which has only one hop left to its end-destination while a multihop PDU is a PDU that has more than one hop left to its end-destination. Figure 3.3 depicts a chain of three nodes where the multihop PDU uses passive/implicit acknowledgement on the A->B link, while the "last hop" PDU uses unacknowledged transmission at the 3a layer and acknowledged transmission at the Logical Link Control (LLC) layer. The implementation of an implicit acknowledgement scheme demands a *Global Identifier* (GId) field in the 3a PCI: a unique identifier that identifies a PDU during its lifetime in the network.

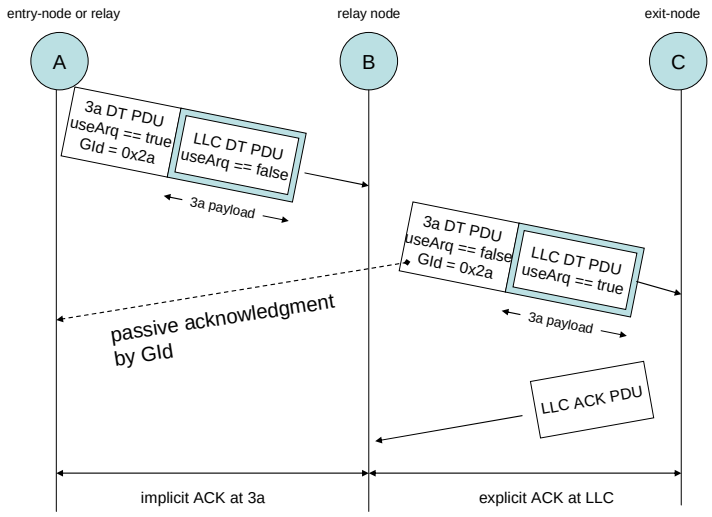


Figure 3.3 The 3a layer protocol utilises the broadcast feature of radio communications by using passive acknowledgement on the intermediate hops and explicit acknowledgement on the last hop.

[14] The literature also uses the terms implicit acknowledgment and echo acknowledgment.

GId = <endDest><endSrc><dataUnitId> and is included in all 3a-DT-PDUs, see Figure 3.4. The GId is also utilised by other protocol functions within the 3a layer.

```
message 3aDtPDU
{
fields:
  // Layer PCI
       int endDest;      // terminal destination address
       int endSrc;       // terminal source address
       int dataUnitId;   // Unit identifier
       int sapId;        // TCP,UDP
       int priority;
       bool useArq;
       int noOfSegments;
       int segmentSeqNo;
}
```

Figure 3.4 Format of the 3a data PDU.

```
message 3aAckPDU
{
fields:
  // Layer PCI
       int endDest;      // terminal destination address
       int endSrc;       // terminal source address
       int dataUnitId;   // Unit identifier
}
```

Figure 3.5 Format of the 3a explicit PDU (needed by a relay node when the 3a source misses the passive ACK and retransmits).

Duplicate Filtering

PDUs may be duplicated within the network due to loss of 3a level acknowledgements, retransmissions and selection of alternative routes. The entry-node assigns a GId to each PDU and all relay nodes stores the GId in a cash/database. If a new PDU with the same GId arrives, this PDU will be deleted since it is regarded as a duplicate.

Segmentation and Reassembly

The entry-node splits the packets received from the terminal into packets of a size acceptable for the LLC layer, and the exit-node reassembles all packets to their original size before sending them to the terminal. All the packet segments are relayed as independent packets.

Flow Control

The node buffer system is scaled to have a small buffer space below layer 3a, and the 3a layer entities can therefore effectively choke the outgoing traffic. The flow control mechanism implemented is described in [9] and an overview is given here by means of Figure 3.6. As the first rule (single-threading), the 3a protocol does not allow more than one outstanding data packet to each of its neighbours. This is achieved by adding a forced idle period (pacing) after transmitting packet 1 in the figure. B starts to relay A's data packet at t_2 and A should obviously defer further transmissions until the passive ACK B->A is received. If A sends in the interval $\langle t_4, t_5 \rangle$ then A interferes with the ACK C->B and reduces the likelihood of successful forwarding

of its own packet. Node A shall sustain from further transmissions to B until a pacing period has elapsed. The 3a protocol entity measures the forwarding delay to each of its neighbours and uses this estimate to set a pacing interval. 3a DT PDUs not requesting ARQ are not subject to this flow control mechanism.

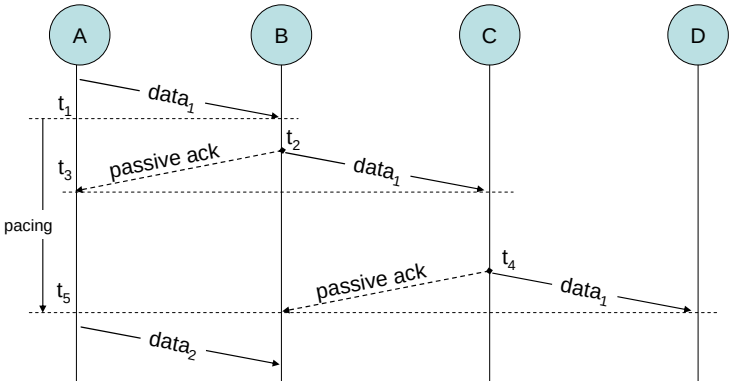


Figure 3.6 Time-sequence diagram for packet forwarding. Related data packets are tagged with the same number (GId).

The 3a layer uses the complex queue structure depicted in Figure 3.7 (class *L3_3aPDP::Buffer*). Queuing of relay traffic and traffic from the local terminal is done within this layer, and this layer determines the serving policy of fresh traffic and transit traffic. The 3a layer can store a large amount of packets in contrast to the LLC layer buffer system.

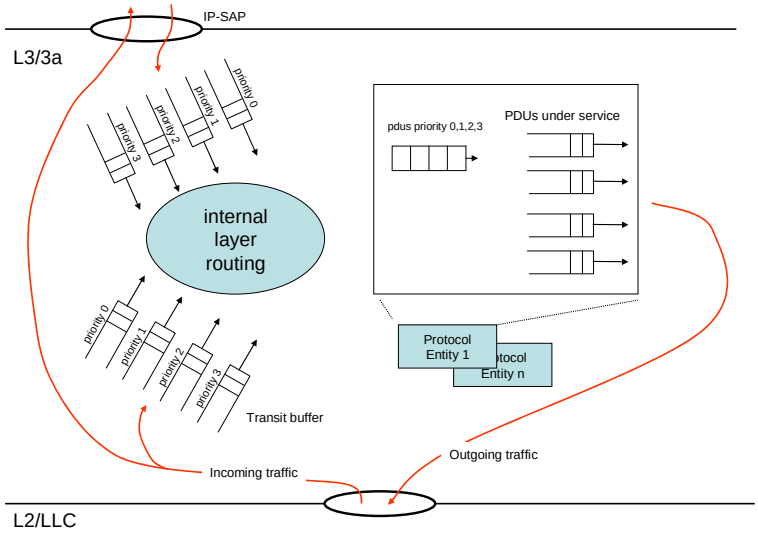


Figure 3.7 The queue structure within 3a layer. Queuing is done on a per PDU basis according to its type and priority level.

3.2 The LLC Layer

The LLC protocol has similar functionality as standard link protocols [19] but is enhanced with some services. The following protocol functions are implemented:

- Data transmission with ARQ using a selective repeat protocol with window size 2
- Data transmission without ARQ
- Lifetime control (is described in section 3.6)
- Precedence and preemption (is described in section 3.5)

The LLC protocol uses the retransmission time delay $t_{Ack} \cdot (2 + IntUniform[0, n_{tx}])$ where the length of the acknowledgement in seconds is t_{Ack}, and n_{tx} is the number of times a packet is transmitted. As explained in section 3.3, *oTWLAN* implements immediate ACK at layer 2 and a feedback should be received within a short time limit. The random component added contributes to a decreased collision rate in the network.

The LLC protocol uses the DT PDU in Figure 3.8 to carry data traffic and the ARQ protocol function uses the ACK PDU in Figure 3.9. The remaining lifetime field is included in the LLC PCI and not in the MAC PCI because the LLC-ACK-PDU does not need a lifetime control function.

```
message LlcDtPDU      // (n)-layer peer-to-peer data
{
fields:
  // Layer PCI
      int constTypeBit   = 0;
      int nS;                 // 2 bits Send sequence number
      int nR;                 // 2 bits Receive sequence number
      bool outOfSeqIndicator;
      bool useArq;
      double remainingLifetime; // [seconds]
}
```

Figure 3.8 The format of the LLC data PDU.

```
message LlcAckPDU        // (n)-layer peer-to-peer data
{
fields:
  // Layer PCI
      int constTypeBit = 1;
      int nR;                 // 2 bits Receive sequence number
}
```

Figure 3.9 The format of the explicit acknowledgement packet used by the LLC protocol.

The queue structure within the LLC layer is shown in Figure 3.10. Identical with the 3a layer, queuing is done on a per PDU basis according to the priority level but here each LLC entity can store one and only one fresh DT PDU per priority. PDUs under service are outgoing acknowledgments, packets scheduled for transmission/retransmission and packets awaiting acknowledgement. *oTWLAN* has no buffers within the MAC layer.

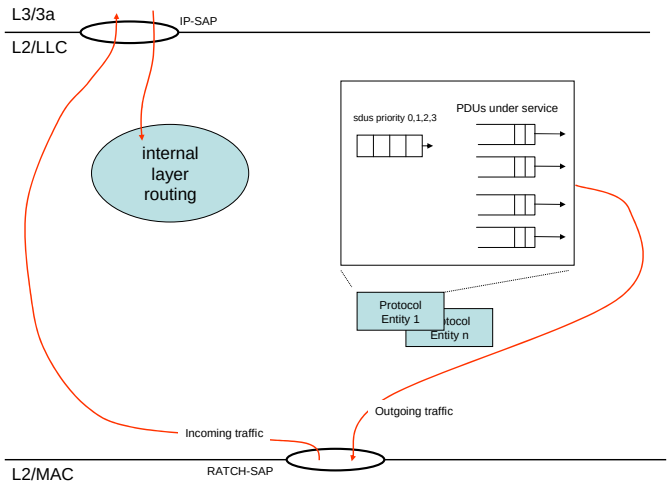


Figure 3.10 The queue structure within the LLC layer. Each LLC entity can store only one (1) fresh PDU per priority. PDUs under service are PDUs ready to be sent (acknowledgments or retransmissions), or PDUs awaiting acknowledgment.

3.3 The MAC Layer

oTWLAN uses a preamble sense random access Medium Access Control (MAC) protocol. Preamble sense means that determination of the radio channel state is based on detecting a radio preamble[15]. If a radio fails to detect the preamble then it acts as if the channel is idle. Figure 3.11 illustrates the channel access cycle. When an ongoing transmission ends, all nodes which have data packets ready for service draw a random access delay. The radio channel remains idle for a random time period C_I (the channel idle period) and becomes occupied when the first node transmits. A listening node detects a busy channel t_v seconds after the start of the first transmission. The length of the channel busy period is C_B. Both C_I and C_B are functions of the offered traffic.

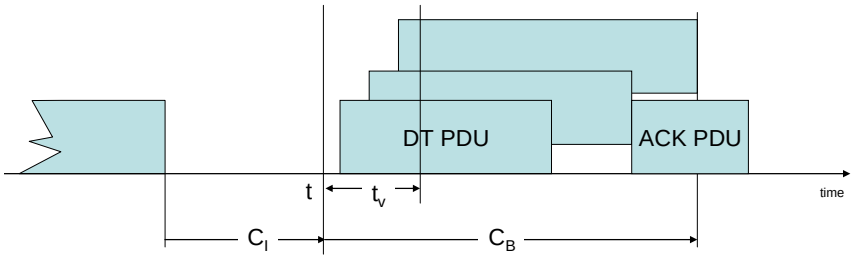


Figure 3.11 The contention phase and the transmission phase of a channel access cycle. The first node transmits at time instance t and a listening node detects a busy channel at t + t_v. The system uses immediate acknowledgment which means an ACK is returned without any delay and may collide with on ongoing data transmission (packets may have different lengths).

[15] Reference [12] outlines channel sense schemes based on energy detectors, preamble detectors and decorrelation-based detectors.

The MAC protocol uses an access delay function *D*, which is the sum of a fixed component and a random component, given by:

$$D(a_p, b_p) = a_p \cdot t_v + RandUniform[0, b_p \cdot t_v],\ a_p, b_p : \text{Integer} > 0,\ p = 0...3$$

where a_p is named the *priority delay* access factor, t_v is the *vulnerable period* and b_p determines the upper bound of the uniformly distributed random delay. t_v is a function of the radio parameters and section 3.4 specifies numerical values.

The MAC protocol handles four priority levels where the acknowledgement packet has the highest priority and is scheduled for transmission using $D = 0$. If a network uses a single priority level then the probability of having a collision with this access delay access function is given by [1, section 5.3.3]:

$$p_{coll} = 1 - (1 - 1/b_p)^n, \quad b_p \geq 1 \tag{3.1}$$

Given exactly *n* busy nodes at the end of the busy period, the mean channel idle period can be expressed as [1]:

$$E[C_I] = t_v b_p / (n+1) + t_v a_p \tag{3.2}$$

The idle period is wasted channel capacity and should be zero. $E[C_I]$ decreases with decreasing b_p but the penalty is increased collision rate, and we realise that there exists a b_p value which optimise the network throughput under a given set of conditions. Chapter 9 considers network throughput as function of a_p and b_p, and proposes a set of values.

The MAC protocol uses the DT PDU in Figure 3.12. No buffering of data is done within the MAC layer.

```
message MacDtPDU      // (n)-layer peer-to-peer data
{
fields:
  // Layer PCI
       int destAddress;  // Mac level destination address
       int srcAddress;   // Mac level source address
       int crc32;        // 4 bytes CRC
}
```

Figure 3.12 The format of the MAC DT PDU.

3.4 The Physical Layer

oTWLAN uses a Direct Sequence Spread Spectrum (DSSS) technique and an air frame format similar to the IEEE 802.11 air frame, see Figure 3.13, where the air frame starts with a preamble. A Physical Layer Convergence Protocol (PLCP) carries a field named F1, which, among others,

informs the receiver about the payload rate. Some military radio systems implement countermeasure against jamming by changing the preamble periodically. They will typically include preamble synchronisation and encryption information in F1. IEEE 802.11 uses a fixed preamble and the vulnerability of using a fixed preamble is discussed in [10]. The receiver is informed about the length of the air frame from a Length Indicator (LI) field and detects bit errors in the PLCP header through a 16-bits CRC check. The preamble and F1 are sent as 256 and 128 chips, respectively, to give a more robust radio channel.

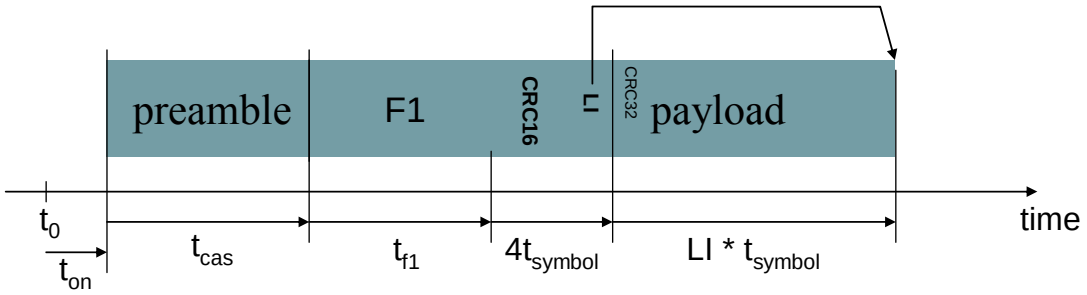


Figure 3.13 oTWLAN air frame format.

oTWLAN implements three different radios, all using the same air frame format, but supporting different payload transmission rates as specified in Table 3.1. The Symbol Error Rate (SER) model used is based on 256-ary orthogonal non-coherent modulation [21] and the simulator implements a separate probability distribution for the preamble, F1 and the payload (class *RadioSER*), see appendix A.

		100kbps	**1Mbps**	**10Mbps**	
$f_{payload}$	Payload transmission rate	0.1	1	10	Mbits/s
f_{chip}	Chiprate	0.4	4	40	MHz
n_{chip}	Number of chips in a symbol	32	32	32	
t_{symbol}	The length of a symbol (8 bits) n_{chip}/f_{chip}	80	8	0.8	μs
t_{cas}	Preamble length $256/f_{chip}$	640	64	6.4	μs
t_{f1}	Symbol length $128/f_{chip}$	320	32	3.2	μs
t_{on}	Receive to transmit switching delay	500	50	5	μs
t_{off}	Transmit to receive switching delay	0	0	0	μs
n_0	Thermal noise density of the receiver	-173.8	-173.8	-173.8	dBm/Hz
n_t	Thermal noise of the receiver	-117.8	-107.8	- 97.8	dBm
$max_{payload}$	Maximum payload size	500	500	500	bytes
$t_{backoff}$	Backoff delay after CRC error or transmission of data: $t_{cas} + t_{f1} + (4+max_{payload})*t_{symbol}$	41.8	4.18	0.418	ms
f_{RF}	Operating RF frequency	50	50	50	MHz

Table 3.1 Physical layer parameters

If a DSSS radio does not receive the preamble correctly, it fails to detect a busy channel. This fact may lead to an instable radio system when using a random access MAC protocol. To guard

against instability, each radio node backs off a period of time after detecting a CRC error and after completing a transmission.

Table 3.1 states an unrealistic operating RF frequency. The purpose of *oTWLAN* is not to provide a tool by which researchers can estimate performance in a real terrain, and the simulator sets the RF frequency to a fixed value (class *UTL_RadioCoveragePredictions*) regardless of the chip rate. We use the default setting 50MHz for the three radio versions. Then they will have the same RF conditions and the performance differences are caused by the radio parameters only.

Section 3.3 stated the importance of having a short vulnerable period (t_v), and it is obviously important to be able to detect the preamble under poor SNR conditions. $t_v = t_{on} + t_{cas}$ can be calculated from the parameters in Table 3.1. The first line in Table 3.2 expresses t_v-values for the three radio versions.

A successful packet reception occurs only when all parts of the air frame are received correctly. Table 3.2 extracts some data from the radio probability distributions to give an indication of the signal level required for successful packet reception. The radio data can be modified by editing the classes *RadioSER* and *RadioConstants* but then you must also modify the parameters for the upper layer protocols.

The simulator provides two other packet capture models described in section 6.7, which are very useful during performance studies.

		100kbps	**1Mbps**	**10Mps**
t_v	vulnerability period	1140 μs	114 μs	11.4 μs
p_{cas}	Probability of preamble detection	0.9 @ - 124 dBm 0.1 @ - 127 dBm	0.9 @ - 114 dBm 0.1 @ - 117 dBm	0.9 @ - 104 dBm 0.1 @ - 107 dBm
p_{F1}	Probability of F1 detection	0.95 @ - 127 dBm 0.4 @ - 131 dBm	0.95 @ - 117 dBm 0.4 @ - 121 dBm	0.95 @ - 107 dBm 0.4 @ - 111 dBm
SER	Symbol Error Rate	10^{-3} @ - 118 dBm 0.6 @ - 124 dBm	10^{-3} @ - 108 dBm 0.6 @ - 114 dBm	10^{-3} @ -98 dBm 0.6 @ - 104 dBm

Table 3.2 Radio performance parameters.

3.5 Priority Handling

In the past, Multi-Level Precedence and Preemption (MLPP) has been a mandatory service in connection oriented military networks [17, 18]. With the introduction of IP terminals, we have no connections to disconnect towards the terminal equipment so preemption in this case must mean to control traffic streams. MLPP shall specify the importance of the *information content* and not the application type. In the simulator, an MLPP value is assigned on a per packet basis at OSI layer 7, and four levels are supported. The MLPP function handles the traffic strictly after rank.

Precedence is needed on data traffic within all layers. Layer 3a and LLC precedence is simply to serve the priority queues in the correct order, see Figure 3.7. The simulator has a very large buffer space and never needs delete data packets for allocation of buffers for higher priority traffic. The MAC preemption process is to interrupt an ongoing MAC scheduling of a lower priority data packet when a higher priority data packet arrives.

The MLPP process at the MAC level has the highest impact on the performance of the MLPP service since it is the MAC protocol that assigns transmission capacity. Implementation of MLPP internally in a node is easy. The difficult part is to construct a mechanism that operates efficiently between network nodes because distributed nodes have no exact and timely information about the queue status in adjacent nodes.

3.6 Lifetime Control

One purpose of packet lifetime control is to stop serving data which have expired, that is, the data is not useful for the recipient(s). Another purpose is to end looping of packets if the routing protocol fails. We also need to have a maximum packet lifetime in the network to be able to reuse unique identifiers, for example, the GuId-field in section 3.1. Lifetime is not intended to be used as a service to take delay measurements!

The principle of the lifetime control function is that the upper layer in the entry-node sets a maximum lifetime value. The 3a and LLC layer entities in the entry-node measure their internal queuing delays. When an SDU is sent down to the lower layer, the remaining lifetime and the internal layer delay are also sent over. Based on these two values, the lower level entity calculates the remaining lifetime and this value is carried by the outgoing LLC PDU. The receiving nodes execute the lifetime control function according to the same procedure. When the remaining lifetime reaches a predefined threshold (a specific threshold is defined for each layer), the corresponding packet is deleted without further actions. The exception is if a receiving node is an exit-node. In this case the packet is always sent to the local terminal.

4 Modelling a Network of Radios

Modelling and simulation are two distinct, yet complementary activities. *Modelling* is the process of creating a model, while a model is anything to which experiments can be applied in order to answer questions about the system modelled. A *simulator* is any object that implements the model and *simulation* is the process of running the simulator. The objective of this chapter is to **model** a network of radios that can estimate throughput-delay characteristics under different conditions. A later chapter will elaborate on how this model can be implemented in a **simulator**.

A model needs to support different operating environments with respect to usage of the communication services and the radio coverage area (network topology). Thus the model must facilitate deployment of radio nodes within a deployment area, referred to as the playground, and provide functions to set up different traffic generators. An example scene is illustrated by Figure

3.1 where a number of radio nodes deployed on the playground serves terminal traffic on a single shared radio channel.

The top level components of *oTWLAN* are shown in Figure 4.1. The user environment, that is, the usage of the application layer services, is modelled by the *User Environment* (UE) box. The arrows signify message passing and the figure illustrates how generators send messages down to layer 7 within the hosts. A *host* is an abstraction of a radio (OSI layer 1 to 3) with an attached user terminal (OSI layer 4 to 7). No messages flow back to the UE and the incoming traffic streams terminate in layer 7 at the exit-nodes. Notice that the hosts are not connected directly but communicates through a physical transmission medium, which models the RF conditions on the radio channel.

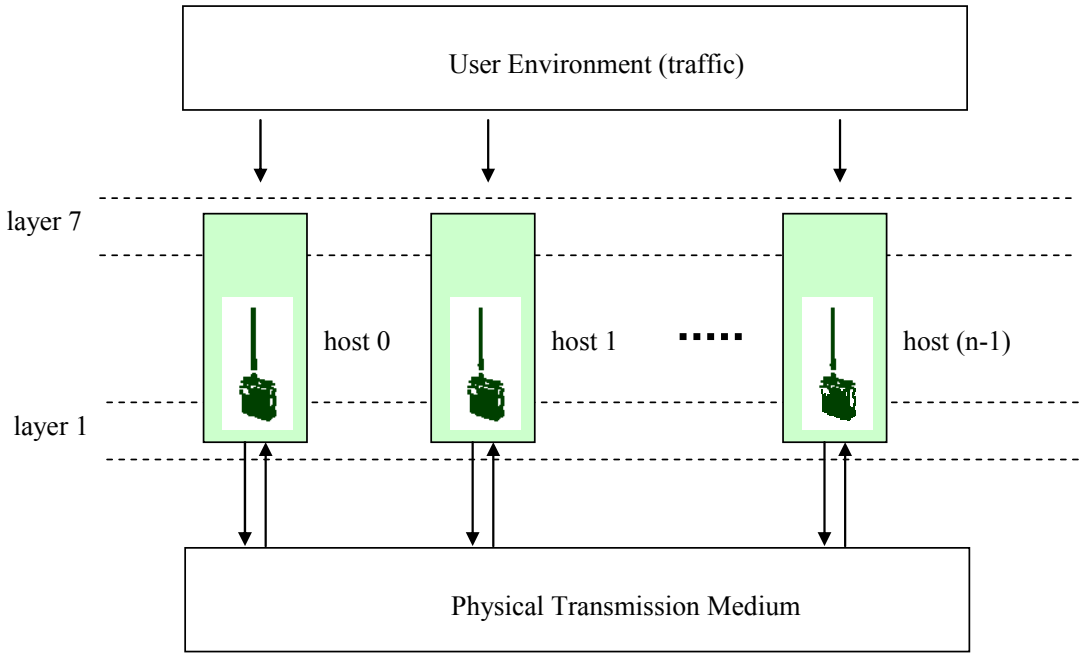


Figure 4.1 The basic structure of the oTWLAN model.

The remaining part of this chapter breaks down the basic structure in a hierarchical manner to end up with a set of sub models of reduced complexity suitable for implementation in a simulator.

The model of a network of radios is a **stochastic discrete event continuous time** model, and has a hierarchical architecture composed of two elementary model types: *atomic* models and *coupled* models. Atomic models are, as the name says, the smallest element of building blocks that are coded manually in some programming language. Coupled models are either composed of other coupled models and/or atomic models.

The top level is a coupled model, identified by the dotted box in Figure 4.2, named *Sim*. *Sim* contains four atomic models and one coupled model, and encapsulates all the other models. *Sim* makes a complete system with any number of hosts.

The coupled model *Host* models a single network node with one radio and one user terminal, that is, a *Host* encompasses layer 1 to 7 of the OSI Reference Model. The model must handle any number of *Host* instances where each host is assigned a unique address (number range is zero to network size minus one).

The atomic model *MChannelControl* models the Radio Frequency (RF) environment of the real world, the physical transmission medium in our basic structure[16]. The functions of the *MChannelControl* is to determine the RF path loss according to the path loss model in use, set the receiving power at the destination end, and copy the RF signals from the *coaxOut* port of a transmitting host to the *coaxIn* port on all the hosts within the radio coverage area of the transmitting host. The *Sim* shall have one instance only of the *MChannelControl.*

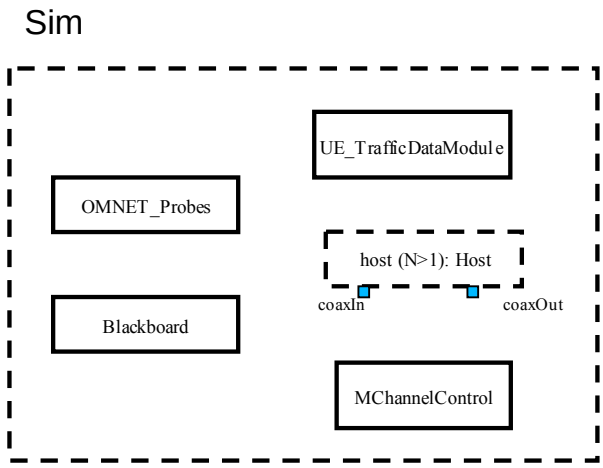


Figure 4.2 The top level of the model is a coupled model containing both atomic and coupled *models.*

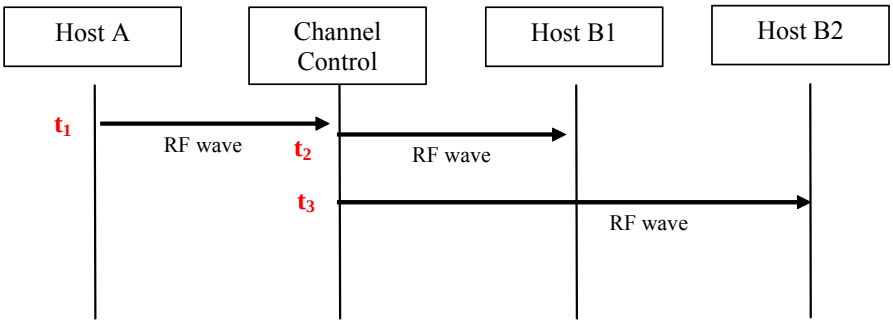


Figure 4.3 Sequence diagram for radio wave propagation when host A transmits. The MChannelControl distributes the wave to all hosts within the radio coverage area of host A.

[16] The Mobility Framework class ChannelControl [11] is taken as the basis for implementing the MChannelControl module (Modified ChannelControl).

```
message RFwave
{
fields:
double txPowerDbm;  // Radiated power in dBm with which this packet is transmitted.
double rxPowerDbm;  // Power in dBm at the receiver antenna input
double rxPowerW;    // rxPowerDbm converted to W
double duration;    // Time it takes to transmit the packet, in seconds!
double startedAt;   // The time instance the wave reached the receiver antenna.
int srcNode;        // The host identifier (>= 0) that sends this packet.
};
```

Figure 4.4 The attributes of the RF wave.

The atomic model *Blackboard* does not exist in the real world, but is included in the model for publishing of cross layer information. Information published on the *Blackboard* does not traverse the radio channel, does not affect the network performance and *Sim* shall have one instance only of the *Blackboard.*

One usage of the *Blackboard* is in conjunction with routing. The *MChannelControl* publishes the link cost matrix (see section 5.4) on the blackboard at time instance zero (the model does not support mobility). By reading this set, the network routing algorithm gets information about radio link connectivity, which is needed to route traffic over multiple radio hops. Another usage of the *Blackboard* may be to implement an address map between internal addresses, range is 0...(n-1), to external addresses (e.g. IP addresses, map[host0].ipAddress gives 127.0.0.8).

The atomic model *OMNET_Probe* collects samples from network distributions (packet loss, queuing delays, etc.) and performs data analysis in run-time. Such functionality would also be needed if network statistics should be measured in a real system. *Sim* shall have one instance only of the *OMNET_Probe.*

An objected oriented approach has been followed during development of a model - there is a one-to-one mapping between the real-life components and the components in the model. A host (layer 1 to 7) in the network contains a network node (layer 1 to 3) and this again contains a *Network Interface Card* (NIC) (layer 1 and 2). A network may contain any number of hosts, but **a host can contain one NIC only**. Motivated by these observations, we introduce a set of atomic and coupled models as shown in Figure 4.5.

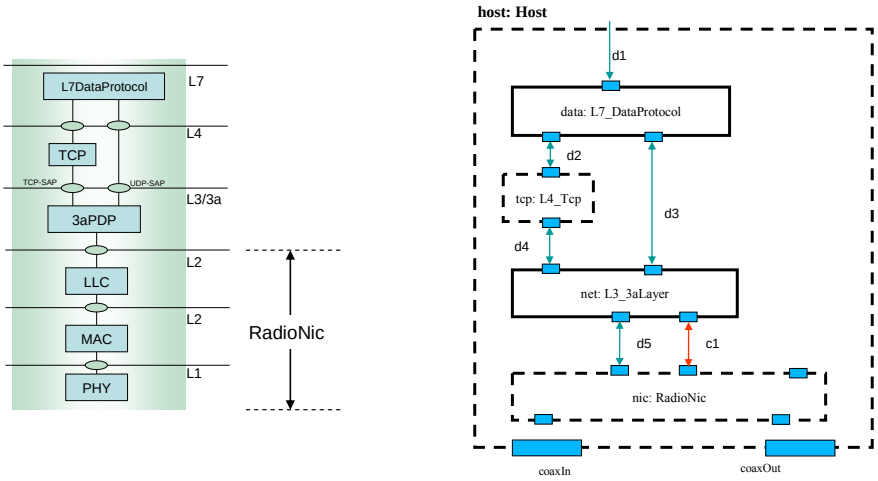


Figure 4.5 The layered reference model (left) and the interior of the coupled model Host (right). Coupled models are marked as dotted rectangles.

The atomic model *L7_DataProtocol* is a simple application layer protocol which handles TCP and UDP traffic. TCP is implemented by the coupled model named *L4_Tcp* in Figure 4.5 and is specified in [3]. Figure 4.6 exhibits its internal organisation.

All the model gates in *oTWLAN* are assigned unique identifiers. Gates used for data traffic are prefixed by the letter "d" in their names while gate names prefixed by the letter "c" carries control messages. The letter is followed by a unique positive integer. Control messages are for coordination of (n+1)-layer and (n)-layer entities, for example, Xon/Xoff flow control between the 3a layer and the LLC layer.

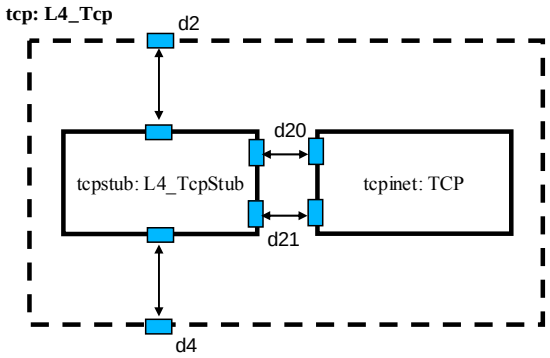


Figure 4.6 The coupled model L4_Tcp contains the atomic module TCP which is implemented by the INET-project [1]. The atomic module L4_TcpStub is designed to be a wrapper between the INET software and the oTWLAN software.

Figure 4.5 above introduced the coupled model *RadioNic* and Figure 4.7 below expands the coupled model to three new atomic models.

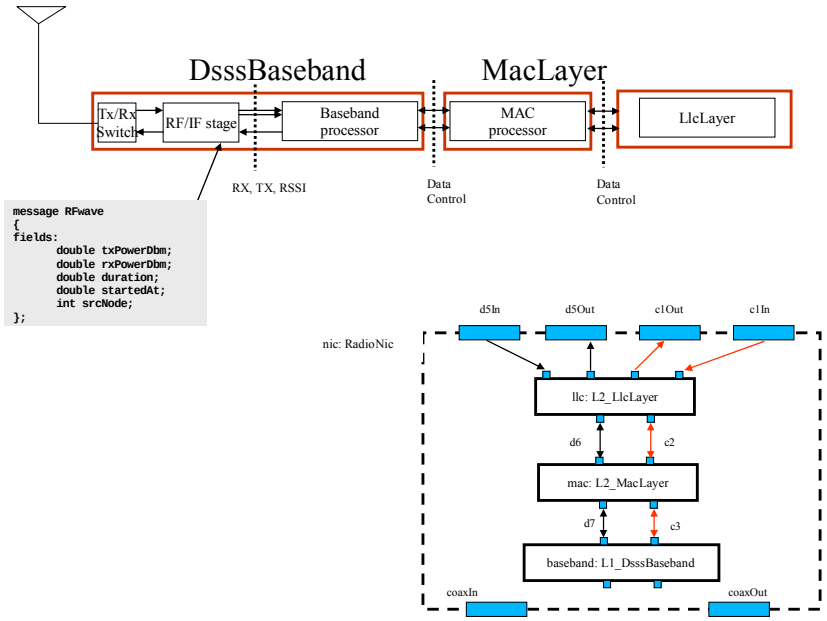


Figure 4.7 The topmost figure depicts an RF wave that arrives at the receiver and shows the internal structure of a radio card. Functional split of the RadioNic into atomic modules is shown in the bottom figure. Incoming RF waves arrive at the coaxIn port. A radio transmission leads to an outgoing RF wave on the coaxOut port.

The atomic model *L1_DsssBaseband* models the DSSS radio while the atomic model *L2_MacLayer* implements the MAC protocol. The *L2_LlcLayer* implements the Logical Link Control (LLC) protocol. Seen from the host point of view, only a well defined interface to the *NIC* is needed and therefore we encapsulated the *DsssBaseband*, *MacLayer* and *LlcLayer* into a coupled model named *RadioNic*. The *RadioNic* provides the two data ports *d5In* and *d5Out*. *d5In* is for layer 3 data from the host that shall be sent over the air interface. *d5Out* is the output port of data received over the air interface. In addition to these two data ports, we have two ports for host internal signalling; *c1Out* and *c1In*. Of course, the *DsssBaseband* and the *MacLayer* must also communicate - both data and local control signalling are needed. This is done over a number of gates as illustrated by the figure.

The modelling hierarchy is now completed since all coupled models are described by atomic models. The *oTWLAN* model is a model of models organised in a hierarchical manner as illustrated by the tree structure in Figure 4.8. The leaf nodes are the atomic models. The top node is the *Sim*, which is a coupled model, and the model may have any number of the nodes named *Host*.

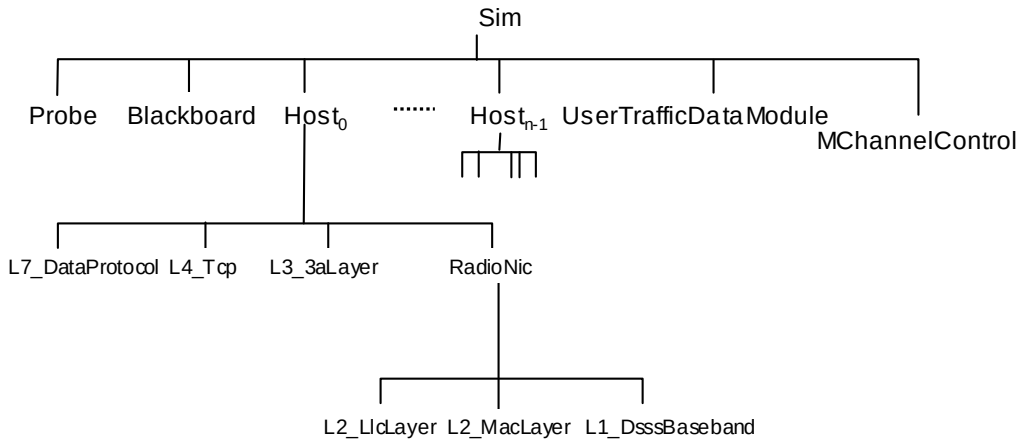


Figure 4.8 The oTWLAN model is a model of models organised in a hierarchical manner.

Before leaving this chapter, we will give a more detailed description of the *MChannelControl* module by means of Figure 4.9. When a transmitting radio sends, the *DsssBaseband* module creates an RF wave and fills in the parameters: srcNode, txPower [dBm], duration [sec], startedAt [sec]. Time values refer to the simulator's time axis. The RF wave message is sent to the *MChannelControl* module, which calculates the signal level for each possible destination, makes an explicit copy of the RF wave to each destination, fills in the rxPower and sends the message to all the receivers. Further processing of the RF wave is done within the receiving *DsssBaseband* module.

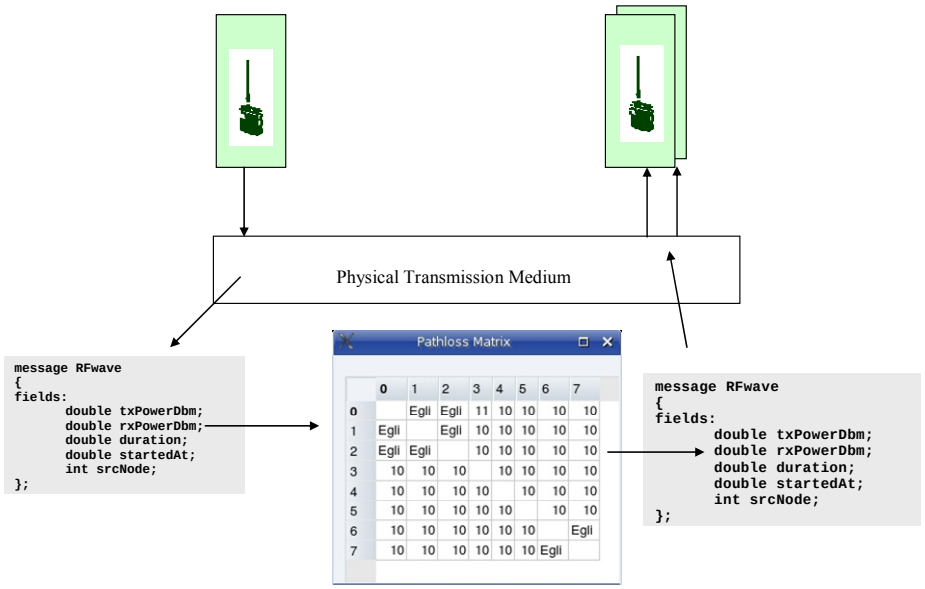


Figure 4.9 The MChannelControl module models the physical transmission medium.

5 Input Data Structures

The purpose of this chapter is to specify the semantics of the simulator's input data structures and give some guidance how they are created by the GUI editors. We refer to the *oTWLAN* doxygen based documentation for a complete description of the syntax.

An ad-hoc network might be considered as having the three planes shown in Figure 5.1. At the bottom we have the playground area - the geographical area where the mobile nodes move around. The next upper plane is the pathloss layer that models the radio wave propagation. The topmost layer is the traffic plane. These planes model the environment with respect to user mobility, the users' deployment location within the terrain and the usage of the communications services (traffic).

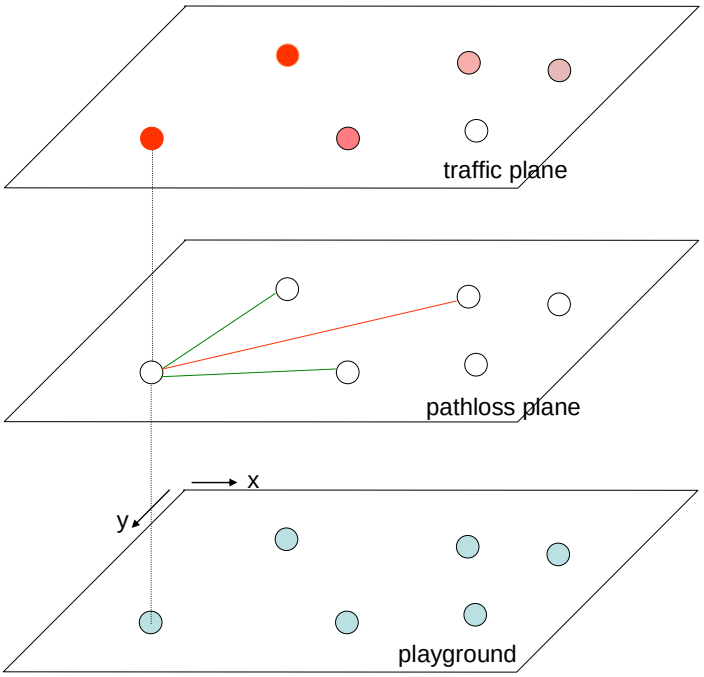


Figure 5.1 Modelling the network environment. These layers must be configured in sequence, starting at the bottom layer. The nodes are assigned unique identifiers within the playground plane and these identifiers are used within the other two planes.

A graph is a collection of vertices and edges. The term *network topology* addresses the graph that represents the connectivity of the network as seen by the network layer. The network topology of *oTWLAN* is, among others, determined by the following models:

The node mobility model
A mobility model defines the spatial distribution of the nodes within the deployed area. A spatial distribution that is time variant represents node mobility.

The pathloss model

A pathloss model defines the RF attenuation function of a radio link. The simulator provides a pathloss model based on Egli [9]. Under this model, the RF condition is affected by the spatial distribution. We need the possibility of having perfect control of the pathloss during the validation of the simulator and the study of protocol behaviour. A fixed pathloss model [17] is therefore provided and this pathloss model is not affected by the spatial distribution.

The traffic model

A traffic model defines the user input traffic to the system (OSI layer 7) in terms of arrival rate and packet length. A traffic generator is characterised by the following attributes:

- Arrival rate distribution [packets/s]
- Payload length distribution [bytes]
- Traffic pattern distribution
- Quality of service: the ARQ distribution[18], priority distribution and packet lifetime

Each node in the system is assigned a unique identifier and this identifier is the same within all the three planes. Nodes can only be added at the bottom layer where the node identities are assigned, but they can be deleted within any plane.

The user of the simulator specifies a network scene by inserting values in a number of data structures, and these data structures are located in XML-based files assigned unique names under the *setup*-directory. The *oTWLAN* dashboard provides GUI based editors for maintaining these files. However, any user may edit the XML-files directly in his favourite editor. The user of the simulator must supply values to these data structures **before** the *oTWLAN* kernel is started. The *oTWLAN* kernel parses the XML-based files when it starts up. The objective of this chapter is to describe the syntax and semantics of the XML-files. Syntax errors or missing data are detected by the Qt XML module while the range checks are done in the *oTWLAN* software. Semantic is important because the *oTWLAN* kernel aborts if some of the rules are broken. Inconsistent data may lead to unpredictable behaviours – the best that can happen is a nice crash with a useful error message. The worst outcome is production of simulation results without detection of erroneous input data!

[17] A symmetric matrix assigns constant loss values.

[18] The ARQ distribution describes the probability of requesting use of ARQ. P(arq)=0 means "no use of ARQ" while P(arq)=1 means that all packets use ARQ.

5.1 Playground

The playground specifies the nodes' positions within the operating area. The playground data graph is specified in Figure 5.2 and the following rules apply for the playground:

1. The playground parameter values cannot change during the simulation session.
2. The playground may define a larger node set "<node>...</node>" than the *networkSize*. This is beneficial since a large playground can easily be scaled down to contain fever nodes by changing a single parameter.
3. The network size determines the number of hosts objects that are created (cf. sim.numHosts).
4. Each host is uniquely identified by an address within the range *0...networksize-1*.
5. If the number of *node*-branches is less than *networkSize*, the playground is invalid.
6. The node vertices must identify exactly one and only one host in the set *0...networksize-1*.
7. There cannot be any missing node addresses in the range *0...networksize-1*.

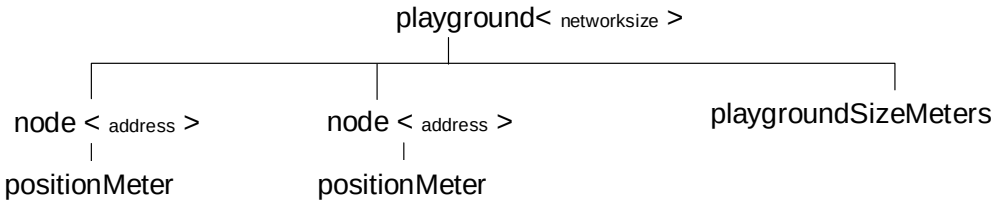


Figure 5.2 The playground data graph is a tree with the top node <playground>.

Here is an example playground of size 5000m X 5000m with two nodes: address 0 and 1.

```
<playground networksize="2" >
  <playgroundSizeMeter x="5000" y="5000" />
  <node address="0" >
    <positionMeter x="200" y="0" />
  </node>
  <node address="1" >
    <positionMeter x="200" y="100" />
  </node>
  <node address="2" >
    <positionMeter x="200" y="200" />
  </node>
</playground>
```

Address 2 is neglected since the address range is 0...1. The playground data structure is implemented by the class *XML_Playground*.

5.2 Pathloss Matrix

The pathloss matrix (class *XML_PathlossMatrix*) specifies the radio link pathloss for all the radio links in the network and its data graph is specified in Figure 5.3. The graph determines the following semantics of the pathloss matrix:

1. The pathloss parameter values cannot change during the simulation session.
2. The default branch specifies default values for all radio links in the network.
3. The node *i* branch specifies deviations from the defaults for the network node identified by address *i.*
4. All the nodes (0...N-1) are first assigned values from the <default>-element and then overwritten by the node *i* branch values.

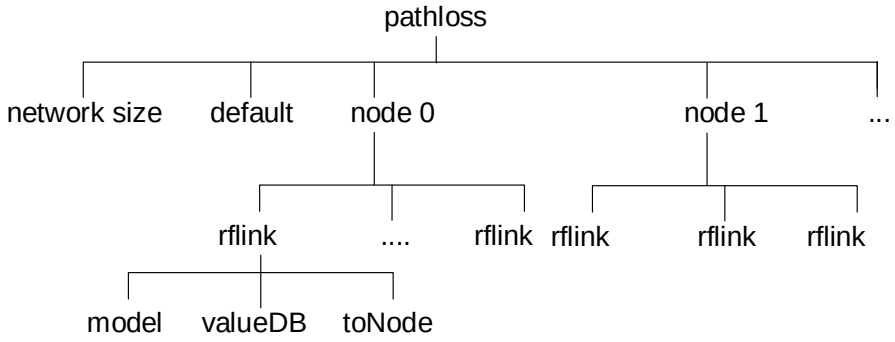


Figure 5.3 The pathloss data graph is a tree with the top node <pathloss>.

The pathloss editor (class *GUI_PathlossEditor*) in *Figure 2.4* builds this data graph after completing the following sequence of stages:

Stage 1 *Set the network size*
Use the spin box to set the network size and the editor creates an *NxN* matrix of radio links where each network node is identified by an integer in the range *0...N-1*.

Stage 2 *Assign a default model*
Assign a default model and default value(s) where needed. The editor appends this default model to all the radio links defined in the previous step.

Step 3 (optional) *Change individual links*
The "Change Links" panel provides functions to overwrite the settings assigned in step 2. You must use the "Add Changes"-button to apply the changes to the matrix. Use the "Show"-button to inspect the matrix.

Step 4 *Save the data*
The editor converts the pathloss data graph to XML and writes it to the file *setup/pathloss.xml*.

The figure below presents an example.

```
<pathloss>
  <size>3</size>
  <default model="Fixed" valueDB="999" />
  <node id="0" >
   <rflinks model="Fixed" valueDB="10" toNode="1" />
  </node>
  <node id="1" >
   <rflinks model="Fixed" valueDB="10" toNode="0" />
   <rflinks model="Fixed" valueDB="10" toNode="2" />
  </node>
  <node id="2" >
   <rflinks model="Fixed" valueDB="10" toNode="1" />
  </node>
 </pathloss>
```

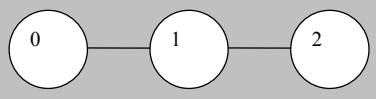


Figure 5.4 This example gives a chain of 3 nodes (since 999 dB pathloss stops all RF waves while 10 dB is a very low attenuation).

5.3 Data Traffic

The simulator's traffic module generates traffic to layer 7 for all the hosts in the system. The traffic specification is stored in an XML coded file and this file can be created manually, or by using the traffic editor (class *GUI_TrafficEditor*). The structure of this file is illustrated in Figure 5.5, and the traffic editor creates vertices in this graph. The graph determines the following semantics of the simulator's traffic (class *XML_UserTraffic*):

1. The network size cannot change during the simulation session.
2. The offered traffic can be specified separately for each run.
3. A run can have any number of generators.
4. One or more generator objects are created for each generator leaf vertex found in the traffic tree (One object for each source in the source address field).
5. The generators' attributes are assigned values in two steps:
   1) Copy in the data found under the default branch; then
   2) Overwrite these data with the data found under the generator branch.

A simulation session must have one or more generators, and a host may have zero or more generators. The effect of having zero generators in a session is that no traffic will be generated in the simulator, and the simulator terminates immediately. The default branch must specify values for all generator attributes, and the generator branch must include at least one attribute.

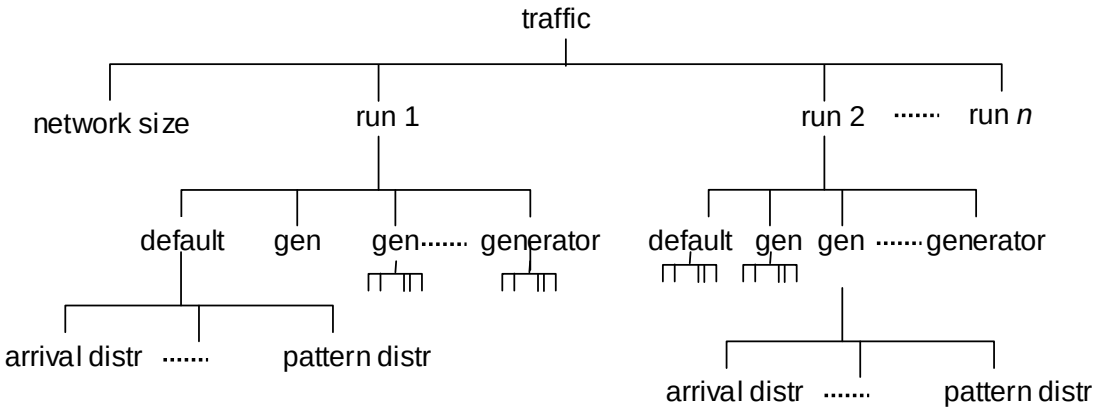


Figure 5.5 The network traffic represented as a graph with the top node <traffic>.

We take an example to clarify the resulting offered traffic per host. Let $g_{\{1,2\}\to\{3,4,5\}}(a)$ denote offered traffic from the random source set $\{1,2\}$ to the random destination set $\{3,4,5\}$ with the arrival rate *a*. The *oTWLAN* kernel creates at run-time two independent generator objects $g_{\{1\}\to\{3,4,5\}}(a/2)$ and $g_{\{2\}\to\{3,4,5\}}(a/2)$ - the L7 packet rates at the two sources are *a/2*. If you specify the two generators $g_{\{1\}\to\{3,4,5\}}(a)$ and $g_{\{1\}\to\{7,8\}}(a)$, the L7 offered traffic from host 1 becomes *2a*.

The traffic editor in Figure 2.12 builds this data graph after completing the following sequence of stages:

Stage 1 *Set the network size*
Use the spin box to set the network size *N* and the simulator creates a container for traffic generators that accepts network node identifiers in the range *0...N-1*.

Stage 2 *Set the default model*
Assign a default model and assign values to its attributes. Then click on the "Set default" to apply the defaults. These defaults are taken as the default values for the generators created in the next step.

Stage 3: *Create the generator(s)*
The previous stage has just assigned defaults and none generator is yet created. Use the "Generator Controls" panel to create traffic generators. Note: The underlying data model is not updated before the "Update"-button is clicked.

Stage 4: *Set the run number*
Use the run number spin box to select the run number and click on "Add run". The generator is added as a new leaf vertex in the correct run branch of the traffic tree.

Step 5 *Save the data*
The editor converts the data graph to XML and writes it to the file *setup/userdatatraffic.xml*.

```
<traffic>
  <networksize size="3" />
  <run number="1" >
   <default>
    <priority P1="0" P2="0" lowestP0="0" highestP3="1" />
    <arrivalDistributionPacketsPerSec model="Fixed" a="0.1" />
    <payloadDistribution model="Fixed" a="50" />
    <lifetime lifetime="60" />
    <arq arq="0" />
    <trafficPattern model="BySrcAndDstSets" from="0" to="2" />
    <protocol use="UDP" />
   </default>
   <generator>
    <arrivalDistributionPacketsPerSec model="Exponential" a="0.01" />
   </generator>
  </run>
</traffic>
```

Figure 5.6 The traffic model for the system is described by an XML-file.

Traffic input data is described by the example XML-file shown in Figure 5.6. Different data apply two different simulation runs, and the <run>-tag identifies the run the data applies to. A generator is first assigned values from the <default>-tag and then overwritten by the attributes defined in its local scope (the area between the <generator>...</generator>). The "from" and "to" attributes reference end-source and end-destination addresses of hosts within the system. Hosts are numbered 0...(<networksize>::size-1). Addresses outside this range lead to modelling errors.

The traffic generators provide fixed and exponential packet arrival distributions with the payload length distributions fixed and uniform. The following traffic pattern models are supported:

From all-to-all.
When a new arrival event occurs (in the *UE_UserTrafficDataModule*), the end-source address and the end-destination address are randomly drawn from the end addresses domain of the system. Routing and relaying are generally needed to serve this traffic type.

From all-to-all RF neighbours.
When a new arrival event occurs (in the *UE_UserTrafficDataModule*), the end-source address *i* is randomly drawn from the addresses domain of system. Then an end-destination is randomly drawn from a RF connectivity matrix belonging to address *i*. Routing and relaying are not needed since the traffic streams terminate in nodes connected directly to the source node.
The RF connectivity matrix is calculated from link cost matrix in Figure 5.7 and the user may set a link cost limit (range $1 \le r \le 1.49$). For example, if the user sets the link cost limit to 1.3, only adjacent neighbours having a link quality better than 1.3 will be included in the RF connectivity matrix.

By source and destination sets.
The *oTWLAN* user sets up explicit source and destination sets before a simulation session starts.

5.4 Routing

The simulator does not implement a routing protocol and the user must specify a routing matrix before the simulation thread starts. This matrix (class *XML_Routing*) specifies the routing path between all combinations of source/destination pairs in the network. Data is stored in an XML coded file named *routing.xml*, and this file can be edited manually using a standard text editor, or created automatically through the routing viewer (class *GUI_Routing*) in Figure 2.10. The routing viewer can build a routing table from a minimum spanning tree based on the link cost described below.

The following rules specify the semantics of the routing matrix:

1. A path must be specified between every pair of source and destination.
2. Addresses are numbered 0...networksize-1.
3. All routing paths are symmetric: path i->j == path j->i.
4. A path is an ordered set where both end-points must be included.
5. The upper triangular must be completely filled in.
6. The routing may be specified as an upper triangular matrix only, and the class *XML_Routing* fills in the lower triangular according to rule 3.

A routing matrix for a 3-node chain looks like this:

```
<routing networksize="3" >
  <path src="0" dst="2" >
    <hopset s="0,1,2" />
  </path>
  <path src="0" dst="1" >
    <hopset s="0,1" />
  </path>
  <path src="1" dst="2" >
    <hopset s="1,2" />
  </path>
</routing>
```

The automatic construction of a routing table is based on the cost-SNR function shown in Figure 5.7. A link having an SNR level greater than b_{snr} is assigned the lowest cost factor one since the link seldom experiences air frame corruption during the low traffic hour (where frame corruption is caused by the background noise only). As the SNR deteriorates, any link reaches a point a_{snr} where the link quality is not sufficiently high for data communications. Links with SNR levels below this value are assigned an infinite cost factor. The (a_{snr}, b_{snr}) parameters are determined by modulation and coding, and *oTWLAN* (see appendix A) uses the threshold values (-6, +3) [dB] (class *UTL_RadioCoveragePredictions*).

Let $SNR_{i \to j}$ [dB] denote the signal-to-noise ratio when node *i* transmits to node *j*, given only one transmission in the network ($SNR_{i \to j} = S_{i \to j} - N_t$). The following algorithm is used for automatic production of the routing table:

Step 1 *Calculate the link cost matrix*

Let C_{rx} be the link cost matrix of order $n \times n$, where *n* is the number of nodes in the network. The pathloss of radio links are symmetric, but since the radios can have different transmitting power, the matrix C_{rx} may become asymmetric. Based on the pathloss model and the radio data, calculate the elements $i, j \mid (i \neq j)$ in C_{rx} using the following link cost function

$$\cos t_{i \to j} = 1 \qquad \text{for } SNR_{i \to j} \geq b$$

$$\cos t_{i \to j} = \frac{0.49}{a-b}(SNR_{i \to j} - b) + 1 \qquad \text{for } a \leq SNR_{i \to j} < b$$

$$\cos t_{i \to j} = \infty \qquad \text{for } SNR_{i \to j} < a$$

The last line sets the radio link to none existing.

Step 2 *Convert C_{rx} to a symmetric matrix*

Loop over all the elements $i, j \mid (i \neq j)$ in C_{rx} : if $c_{i,j} \neq c_{j,i}$ then set $c_{i,j}, c_{j,i} = \max(c_{i,j}, c_{j,i})$.

Network protocols demand bidirectional radio links.

Step 3 *Build the all-pairs shortest-path graph*

An all-pairs shortest-path problem is to find a shortest path from every vertex to every other vertex in a graph. Within our context, this means to take the cost matrix from the previous step and feed it into an algorithm that solves the shortest-path problem.

In principle, this routing model does not exclude mobile hosts - just build new routing tables when one or more hosts have moved a sufficiently long distance to give significant change of the network topology. However, we have already decided to exclude mobile hosts and build the routing table once at time instance zero. The routing table is published on the blackboard and becomes available to all the hosts in the network.

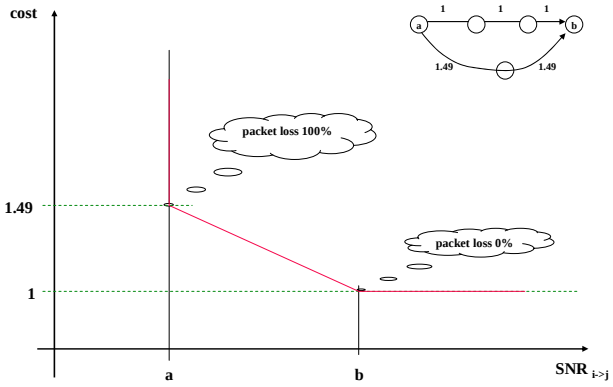


Figure 5.7 The radio links are assigned a cost factor according to the SNR-level. The demodulation threshold cost value is set to 1.49 to force the routing to select the two hops routing path instead of three hops at the upper right corner of the figure.

5.4.1 A case study

The aim of this section is to illustrate the practical aspects of the link cost and usage of the routing viewer (Figure 2.10). Consider the scenario in Figure 5.8 where seven nodes are deployed on the playground.

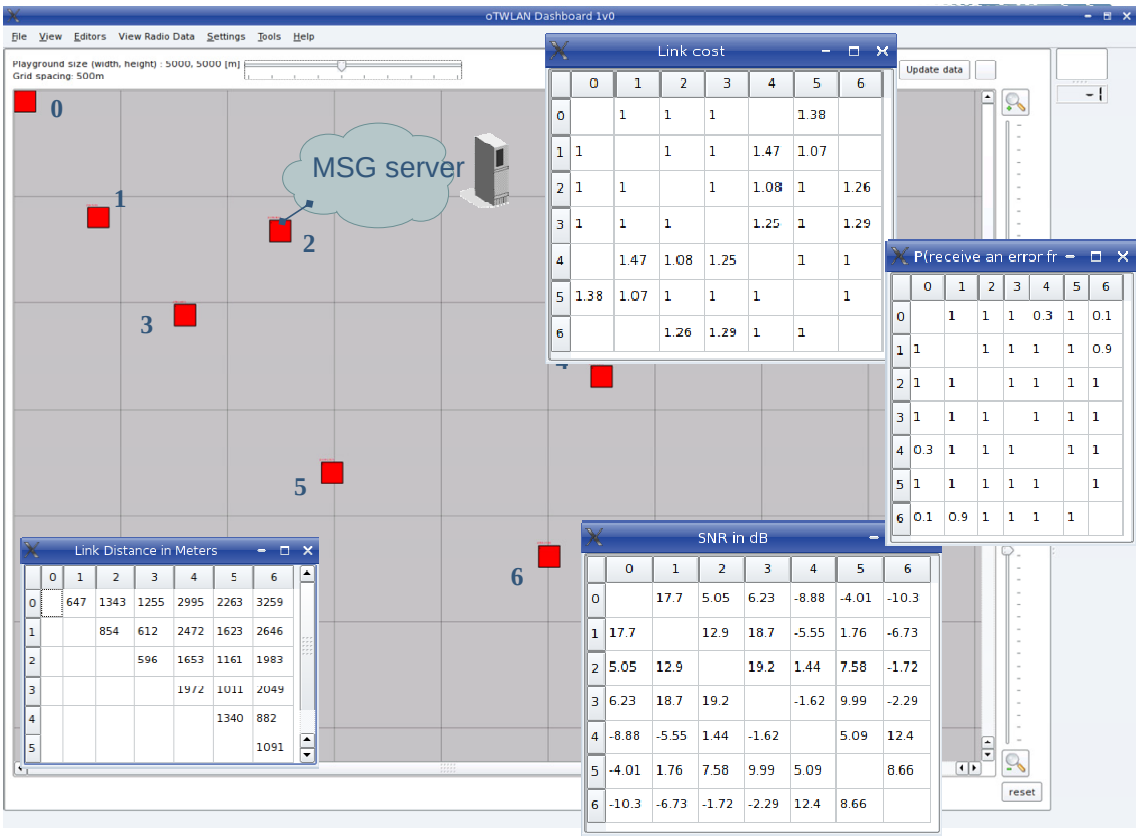


Figure 5.8 A playground with seven nodes.

Node 2 has a central position since it operates as a router between the wireless clients and the infrastructure. The link cost table shows that this is not a fully-connected network. For example, node 0 is not connected to node 4. The SNR table expresses -8.9 dB SNR on this link and demodulation never succeed[19]. However, the preamble detection probability is 0.3 (the right table) and thus node 0 and node 4 are not completely hidden nodes in the sense that the MAC protocol prevents collisions from time to time. The routing viewer in Figure 2.10 displays routing graphs after selecting a top node.

Of particular interest in this scenario are the edge node 0 and the gateway node 2. The output graphs from the routing viewer are presented in Figure 5.9. Node 2 reaches all nodes in one hop and this is beneficial due to its importance for having fast access to the message server. Node 0 may communicate with all the other nodes in the network but the traffic must be relayed over three hops to reach node 4 and node 6.

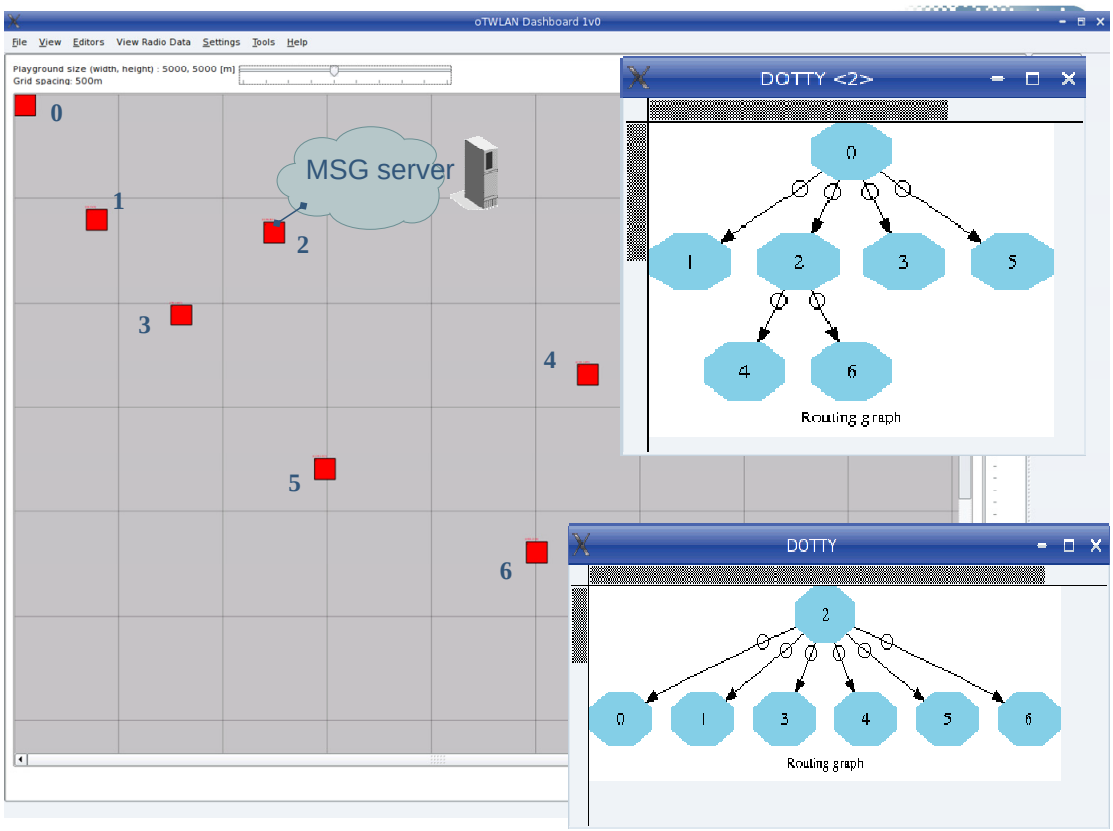


Figure 5.9 The routing graph for node 0 and node 2.

[19] The SNR table applies to the low traffic hour where bit-errors mostly are caused by the background noise. As the traffic increases, the SNR becomes degraded due to collisions.

5.5 Radio Data

Radio data (class *XML_RadioData*) specifies basic physical layer attributes such as power and antenna height, and the data structure has a default section containing these attributes:

```
<txpower watt="1.0" />
<antenna txGainDBi="0.0" rxGainDBi="0.0" heightInMeters="2.0" />
<captureModel model="Normal" backgroundNoise="0.0" />
<equipmentVersion hw="Version1" />
```

The <captureModel>-tag (cf. section 6.7) and the <equipmentVersion>-tag have a global scope in the sense that all radio nodes in the network use the same value. The latter describes the radio hardware in use, different hardware is assumed incompatible, and all the nodes must use the same version number. However, <txpower> and <antenna> can change between nodes. The data graph is shown in Figure 5.10 and the following rules apply:

1. All nodes are assigned data from the <default>-section.
2. When a node address is found in the <node>-section, data assigned from the <default>-section is overwritten by attributes under the <node>-section.
3. The following tags shall not be included in the <node>-section: captureModel and equipmentVersion.
4. The <default>-section is mandatory and all the tags must be present with valid values.
5. The <node>-section is optional.

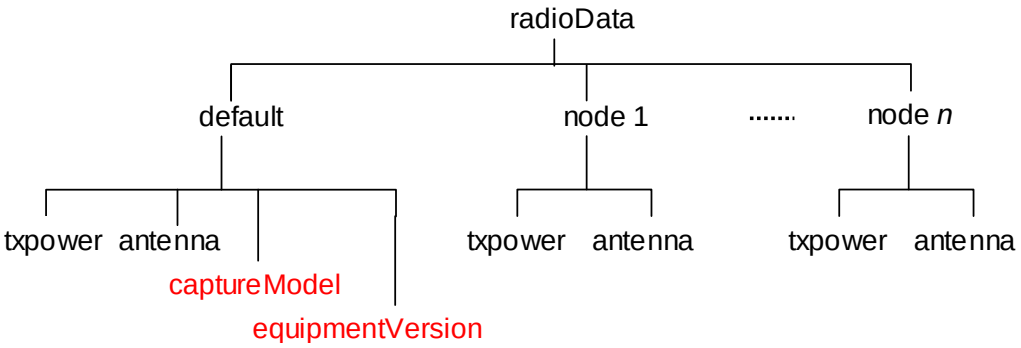


Figure 5.10 The structure of the radio data.

The physical layer editor (class *GUI_PhyLayerEditor*) in Figure 2.5 provides a tool box (cf. QToolBox) where the page “Global Parameters” assigns values to <captureModel> and <equipmentVersion>, while the page “Default parameters” assigns values to the rest of the <default>-section. The usage of the physical layer editor is as follows:

Step 1 Global Parameters
Select the page “Global Parameters” and make your choices.

Step 2 Default Parameters
Select the page “Default parameters” and change the spin box settings.

Step 3 (Optional) Node Parameters
Select the page “Node Parameters” and set values on per node basis.

Step 4 Save the data to the file *setup/radiodata.xml*.

Here is a printout of a radio data XML file:

```
<radioData>
  <default>
    <txpower watt="1.5" />
    <antenna txGainDBi="0.0" rxGainDBi="0.0" heightInMeters="2.0" />
    <captureModel model="Normal" backgroundNoise="0.0" />
    <equipmentVersion hw="Version1" />
  </default>

  <node address="0" >
    <txpower watt="3.0" />
  </node>
  <node address="1" >
    <txpower watt="30.0" />
    <antenna txGainDBi="1.0" rxGainDBi="1.0" heightInMeters="3.0" />
  </node>

</radioData>
```

The *backgroundNoise* element applies only to the capture models “perfect capture” and “zero capture”.

5.6 Probe Data

The probe module [2] takes its input from an XML coded file (named *probeInFile.xml*) and this file specifies which probes shall be activated for the simulation session. Figure 5.11 shows the structure of the file controlling the probe module, and the probe editor creates vertices in this graph.

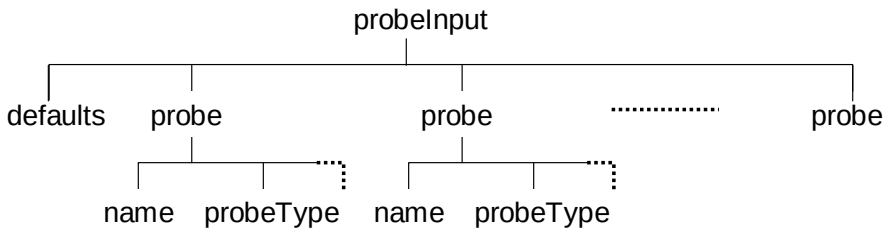


Figure 5.11 The probe input data graph is a tree with the top node <probeInput>.

The graph determines the following semantics of probes:

1. The probe module configuration is the same for all the session run numbers.
2. The default branch may be omitted.
3. A simulation session may have any number of probes.

4. All probes are disabled by default[20] and every probe is uniquely identified by a human readable text string. When a valid name vertex is found in the probe branch, the state of the probe object is switched from disabled to enable.

The probe editor in Figure 2.14 builds this data graph after completing the following sequence of stages:

Stage 1 (optional) Set the default values
Click the button named “Edit defaults” and a panel for editing the probe attributes appears.

Stage 2 Enable/disable probes
Use the “Insert probe” or “Remove probe” to enable or disable probes.

Stage 3 (optional) Change individual probes
Click on the probe name in the probe table widget and a probe attribute editor appears showing the current settings. Change the attributes at will.

Step 5 *Save the data*
The editor converts the data graph to XML and writes it to the file *setup/probeInFile.xml*.

Here is a probe example file:

```
<probeInputFile>
  <defaults>
   <probeType>Terminating</probeType>
   <alfaConfidenceCoefficient>0.9</alfaConfidenceCoefficient>
   <accuracy>0.1</accuracy>
   <trace>false</trace>
   <maxmin>true</maxmin>
   <roLimit>0.4</roLimit>
   <transientPeriodLength>10</transientPeriodLength>
   <windowSize>-1</windowSize>
  </defaults>
  <probe>
   <name>endToEndDelay</name>
   <probeType>SampleMean</probeType>
   <accuracy>0.3</accuracy>
  </probe>
 </probeInputFile>
```

5.6.1 Rate probes

A rate probe takes measurements over a time window and forms a time average at the end of the window. Examples of a rate probes are the offered traffic probe that estimates the number of bytes per second generated by the source terminals, and the throughput probe which measures the number of bytes/s reaching their exit nodes.

[20] Probes are static objects in the simulator. Thus they always exist but are deactivated by default.

Rate probes collect samples over time windows of size t_w seconds and form a time average over the window that is treated as a single sample during data analysis. Figure 5.12 illustrates the algorithm used by the rate probes[21]. The two packets (k) and (k+1) arrive in window (i) while packet (k+2) arrives in window (i+1) and it is the time average over each window that is taken as a sample.

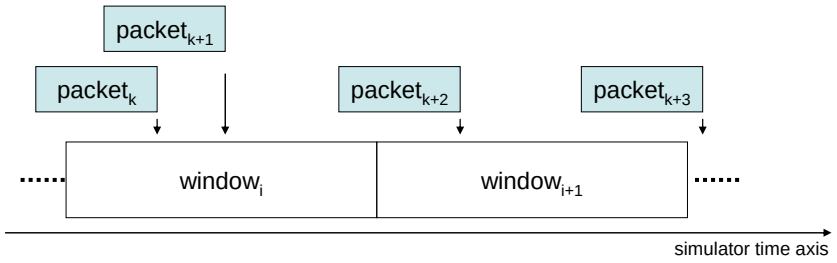


Figure 5.12 Rate probes collect samples over a time window.

When the packet arrival process is a Poisson process with intensity λ packets per second, the expected number of arrivals in each window is $n_w = \lambda \cdot t_w$ (Little's law). If n_w is a small number, we have a high likelihood to have no arrivals during the simulation period and all windows produce a zero and the mean value is estimated to zero. This is clearly not a correct value. If the rate probe is activated as a terminating probe [2, chapter 2], the same situation may occur if $5000 \cdot \lambda t_w$ is a small number, but the consequence is different[22]. Collecting 5000 zeros leads to accuracy = halfWidth/mean = 0/0 = 1, the batch means procedure stays in its estimation state and demand more samples.

From the discussion above we conclude that n_w should be large, but since the packet intensity distribution λ is unknown it is impossible to calculate t_w in advance. For this reason, the probe module has been extended to provide a function which adjusts the time window at run-time to have exactly two samples in each window. The user activates the adaptive window function by inserting the string "auto" instead of inserting a double on the window size input line in Figure 5.13.

[21] This section outlines rate probes because the *oProbe* project [2] does not describe rate probes. *oTWLAN* also extends its functionality.
[22] The batch means procedure needs 5000 samples before data analysis can start.

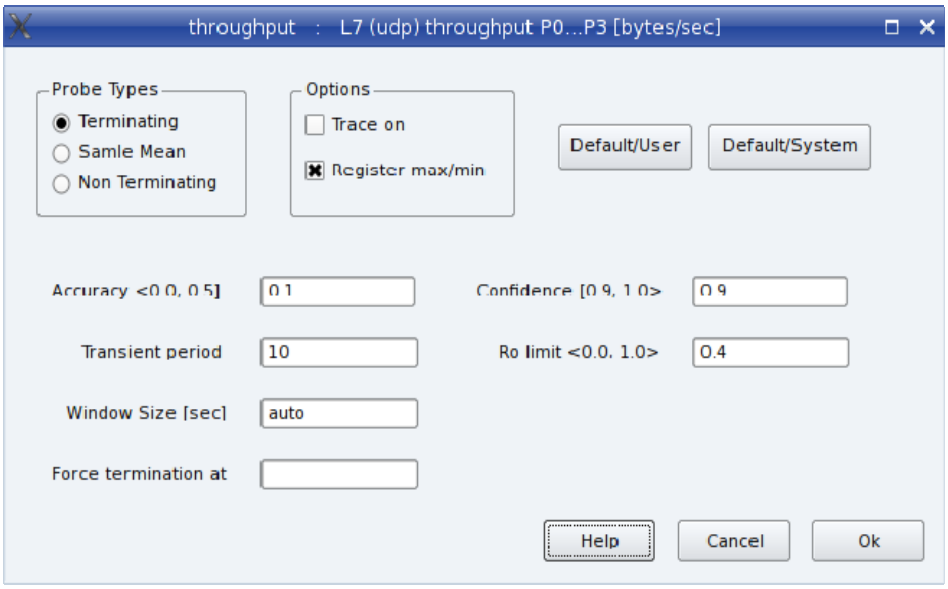


Figure 5.13 The Probe Editor (class oProbe::GUI_ProbeEditor).

A rate probe activated as a sample mean probe can safely use the automatic window size function without experiencing a longer simulation time or producing erroneously results. To study the effect of different t_w-sizes in conjunction with the batch means procedure, we estimated the offered traffic from a single traffic generator using a Poisson arrival process with rate 1 packet/s and 400 bytes fixed packet length. The offered traffic is then known to be 400 bytes/s. Table 5.1 presents the results of these trials under various window sizes.

Window size t_w [sec]	Estimated mean	Sample size	Run time [sec]
10	400.8 ± 3.9	6002 (*10)	5.5
1	400.4 ± 4.2	57502 (*1)	5.3
0.1	400.4 ± 4.3	562502 (*0.1)	7.2
0.05	400.4 ± 4.0	1137502 (*0.05)	12.3
0.01	400.4 ± 4.0	5612502 (*0.01)	148
Automatic	400.5 ± 3.6	30002 (*2)	5.8

Table 5.1 Estimated offered traffic in bytes/s as 99% confidence intervals.

Firstly, note that all confidence intervals cover the true mean (400 bytes/s). The sample size column expresses the number of samples gathered at time of termination. The total number of packets included in each sample is identified by (*x). For example, 6002 (*10) means 6002*10 packets were required before the accuracy was reached. All the runs demanded approximately the same number of packets but the table shows significant deviations in the execution times. This is caused by the increasing event rate generated by the probe module as the observation window decreases. The lessons learned from these experiments are: 1) Use the window size “automatic” since only an insignificant increase in run-time is experienced and you need not to bother about the problem of selecting a sufficient large window size compared to the underlying rate distribution and 2) Turn off all rate probes not needed to prevent unnecessary generation of events.

5.6.2 Probe objects

Probes are the most important tool to get results from simulators and *oTWLAN* implements a large number of probes that estimates the first order moment of the underlying probability distributions. This section explains what the different probes measure. Probes tagged by priority (P0,...,P3) do only sample traffic at the corresponding priority level.

throughput, throughputP0,..., throughputP3 [bytes/s].This is the layer 7 throughput and is the sum of all packets that successfully pass between end terminals. The sampling is done in the *L7_DataProtocol*-module. Five throughput estimators are available; one for each of the four MLPP levels and one sums the throughput over all MLPP levels.

offeredTraffic [bytes/s]. This is the sum of the traffic sent by the user environment module to the hosts. The sampling is done in the *L7_DataProtocol*-module. A god practise is to activate this estimator in all simulations to detect faulty setting of traffic generators.

endToEndDelay, endToEndDelayP0,..., endToEndDelayP3 [seconds]. The end-to-end packet delay is the difference between the time of delivery and the time of arrival measured at layer 7.

3aNReTx. This estimator measures the number of times the layer 3a entity must retransmit packets. Only packets that shall use ARQ are included in the statistics.

3aQueueTimeTransitTraffic [seconds]. The average waiting time for all traffic in the transit buffer (Figure 3.7). Transit traffic is traffic received from the neighbourhood and shall be relayed by this node. Sampling is done when the 3a PDU is sent down to the LLC layer and the PDU may get an additional queuing delay within the LLC layer.

3aQueueTimeLocalTraffic [seconds]. Measures the average waiting time for all traffic received from the layer above layer 3 (Figure 3.7) and represents traffic that have not yet been sent on the radio channel. Sampling is done when the 3a PDU is sent down to the LLC layer and the PDU may get an additional queuing delay within the LLC layer.

3aHopCount. Measures the number of hops the packets traverse end-to-end. You have a means to detect a faulty routing table if you activate sampling of max/min values of this estimator.

llcNReTx. This estimator measures the number of times the LLC entity must retransmit packets. Only packets that shall use ARQ are, of course, sampled.

llcQueueTime [seconds]. The LLC layer may store one fresh LLC SDU per priority and this estimator measures the waiting time in this queue, that is, the time delay until it get served by the LLC entity.

phyNKTx. This estimator measures the number of simultaneous transmissions present when the baseband processor (class *L1_DsssBaseband*) gets an F1 detection alarm. If phyNKTx = 1 then

no collisions have occurred. The estimator takes the average over all nodes in the network. This estimator gives a good indication of the conditions in a well connected network. However, in a fragmented network the MAC protocol is less efficient and collisions may frequently occur after F1. Those events will not be measured by this estimator.

phyECI [seconds]. This estimator measures the channel idle period. Compared to the theoretical channel idle period in Figure 3.11, a radio in the simulator detects a busy channel t_v-seconds later. When a radio detects a CRC error, it may potentially have lost the control of the channel state and samples are only taken when packet corruptions do not occur.

5.6.3 Counters

Counters (class *oProbe::Counters*) are objects of the type positive integer that sum up something, for example, the number of events of a certain kind during a session run. The counters are set to zero at the start of each session run and the results are printed to a file *counters_rN.txt* at the end of run *N*. The "N" (=1,2,3,...) identifies the session run number.

Counters make samples from stochastic distributions. The quality of the measurements is not controlled and therefore counters shall never be included in a simulation report. The practical usages of counters are:

1) Confirming the correctness of your arguments made from the probe output data
2) Discovering erroneous setting of input data
3) Understanding location dependent behaviour

An example of bullet 2) is given in section 7.1. With the assistance of the simulation example in chapter 2, we concretise the meaning of bullet 3). Consider the following counter output for the low traffic hour (session run number 1):

```
sim.host[4].nic.baseband::L1_DsssBaseband:noOfCasAlarms = 18090
sim.host[4].nic.baseband::L1_DsssBaseband:noOfRfWavesRxed = 18090
sim.host[4].nic.baseband::L1_DsssBaseband:noOfF1Errors = 0
sim.host[4].nic.baseband::L1_DsssBaseband:noOfPlcpErrors = 0
sim.host[4].nic.baseband::L1_DsssBaseband:noOfCrc32Errors = 0

sim.host[8].nic.baseband::L1_DsssBaseband:noOfCasAlarms = 18115
sim.host[8].nic.baseband::L1_DsssBaseband:noOfRfWavesRxed = 18115
sim.host[8].nic.baseband::L1_DsssBaseband:noOfF1Errors = 0
sim.host[8].nic.baseband::L1_DsssBaseband:noOfPlcpErrors = 0
sim.host[8].nic.baseband::L1_DsssBaseband:noOfCrc32Errors = 68
```

Note the similarity between the counters' structure and the simulator's data graph in Figure 4.8. The host numbers inserted are identical with the node addresses displayed on the playground. Thus host 8 is the edge node 8 in Figure 2.3, while host 4 is the centre node on the playground. The counter *noOfRfWavesRxed* sums up the number of RF waves reaching the coax input port on the radio. RF waves reach the coax input port even though the radio is in a transmitting state. The output shows that both nodes got a carrier sense alarm (CAS) on each RF wave received. Node 4 never experienced demodulation failure while node 8, which operated under poorer SNR

conditions, failed a number of times. The same output for the high traffic hour (session run 19) follows here:

```
sim.host[4].nic.baseband::L1_DsssBaseband:noOfCasAlarms = 53636
sim.host[4].nic.baseband::L1_DsssBaseband:noOfRfWavesRxed = 58472
sim.host[4].nic.baseband::L1_DsssBaseband:noOfF1Errors = 3
sim.host[4].nic.baseband::L1_DsssBaseband:noOfPlcpErrors = 1078
sim.host[4].nic.baseband::L1_DsssBaseband:noOfCrc32Errors = 513

sim.host[8].nic.baseband::L1_DsssBaseband:noOfCasAlarms = 53280
sim.host[8].nic.baseband::L1_DsssBaseband:noOfRfWavesRxed = 58153
sim.host[8].nic.baseband::L1_DsssBaseband:noOfF1Errors = 16
sim.host[8].nic.baseband::L1_DsssBaseband:noOfPlcpErrors = 1021
sim.host[8].nic.baseband::L1_DsssBaseband:noOfCrc32Errors = 618
```

Here we see an increase in the demodulation failure rate due to collisions.

6 Simulator Design

A simulator is any computer program which implements the model described in chapter 4. To design a simulator means to get a step further towards an implementation. This chapter describes how the *oTWLAN* is designed to run under the *OMNeT++* simulation framework [1]. The user front-end is based on the *Qt* open source project [7].

6.1 Design Patterns

This section presents some general guidelines applied during the design and implementation of *oTWLAN*. The objective is to have general schemes that are applicable for many of the protocol layers. Usage of common patterns makes it easier to write, read and reuse software components.

The atomic models defined earlier are implemented as *OMNeT++* simple modules (class *cSimpleModule*) and the coupled models are implemented as *OMNeT++* compound modules.

Gate names containing the string “c” (for control) are reserved for signalling of local control information within a host, the content shall never be sent over the air interface. Control ports shall never receive PDU/SDU messages. Gate names containing the string “d” (for data) are reserved for PDU/SDU messages that are intended to be sent over the air interface (or to local terminals). Data ports shall never receive local control information.

The message exchange between modules is based on classes which inherit the class *cMessage*. The messages passing between layers follow the principle of the OSI Reference Model where each layer entity adds protocol control information (PCI) for (n)-layer peer-to-peer communication. Local information exchange between the (n)-layer and the (n-1)-layer is passed as Interface Control Information (ICI), as illustrated by Figure 6.1.

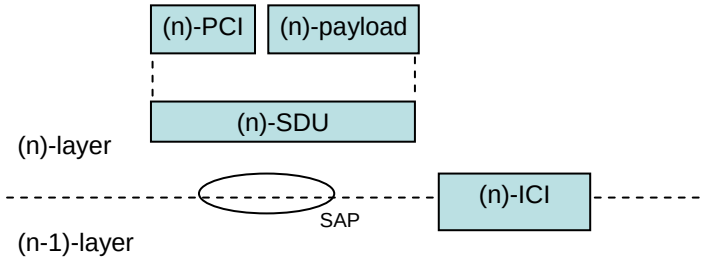


Figure 6.1 General principles for encapsulation and interface control.

The *OMNeT++* framework includes functions supporting encapsulation/decapsulation and addition of interface control information to messages. Therefore, all message exchange in the situations identified by Figure 6.1 shall use the message template in Figure 6.2.

```
enum Constants
{
  OmnIni    = -1;     // means that values are taken from the omnettpp.ini file
  Undefined = -100000;// assign an illegal value (better to crash than
                      // simulate with legal but uncontrolled values)
  StatDbg   = -100001;// variables for statistics and dbg
};

message zzzPDU  // (n)-layer peer-to-peer data
{
fields:
      // Layer PCI
      int constPciLength =...;//The total length of the PDU PCI in #bytes.
      int yourPci1 = Undefined;
}

message zzzyyyNetInterface // (n+1)-layer <-> (n)-layer interface data
{
fields:
      int yourVariable1 = Undefined;
      int yourVariable2 = Undefined;
}
```

Figure 6.2 Templates for layer PDU and ICI. ”zzz” identifies the (n)- layer and ”yyy” identifies the (n-1)-layer. These messages are compiled by the OMNeT++ message compiler which produces two classes. A single variable constPci shall be defined for each PDU to facilitate easy definition of layer overhead per layer in number of bytes.

An important principle for us is to save all model input data together with the output data from the simulation experiments. We are then able to repeat our experiments later - we often do to check, among others, the effect of bug fixes. Figure 6.3 illustrates the basic design scheme applied to fulfil this work principle. All input/output shall be read/saved in XML-files and the interactions with XML-files shall go via DOM trees.

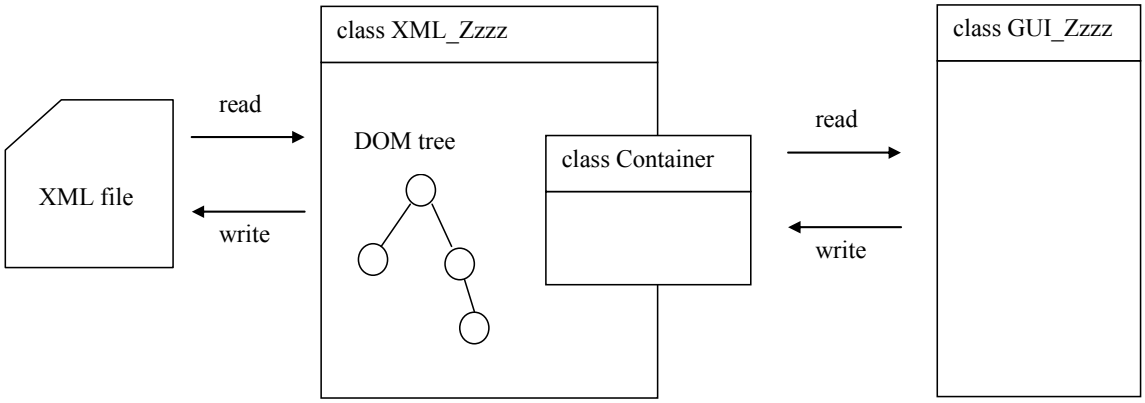


Figure 6.3 Creation and visualisation of XML file content.

6.1.1 Qt4 Based Models and Views

The Model-View-Controller (MVC) pattern described in [6, chapter 17] is used by the User Interface (UI) software package specified in chapter 8. A *model* is the domain specific representation of the information that the application operates. *Views* render the model into a form suitable for interaction with people while the *controller* integrates the model and the view, and decides how the user interface reacts to user input. The controller creates view/model instances and connects them, see Figure 6.4.

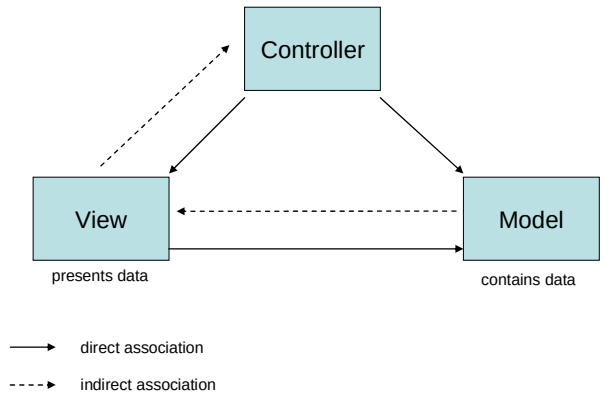


Figure 6.4 Depicting the relationship between controller, view and model.

To concretise the meaning of MVC, we look at the playground editor in Figure 2.3 – one of the first widgets seen by the *oTWLAN* user. The graphical node items on the playground are implemented by the class *GUI_PlaygroundNodeItem::public QGraphicsItem* and the *GUI_PlaygroundScene::public QGraphicsScene* implements the surface for managing node items or other graphical items on the playground. The playground editor implemented by the MVC pattern consists of the three classes (in sequence):

GUI_PlaygroundModel : public QAbstractTableModel
GUI_PlaygroundView : public QGraphicsView
GUI_PlaygroundController : public QObject

Note the base classes which are Qt4 classes.

6.2 The User Traffic Module

The purpose of the user traffic models is to emulate the usage of the application layer services. The atomic model named *UE_UserTrafficDataModule*[23] defined in chapter 4 contains the traffic generators. Notice that the *UE_UserTrafficDataModule* module operates above layer 7.

A data traffic generator is characterised by the following attributes (Figure 6.5):

- Arrival rate distribution [packets/s]
- Payload length distribution
- Traffic pattern distribution
- Quality of service: the ARQ distribution, priority distribution, lifetime

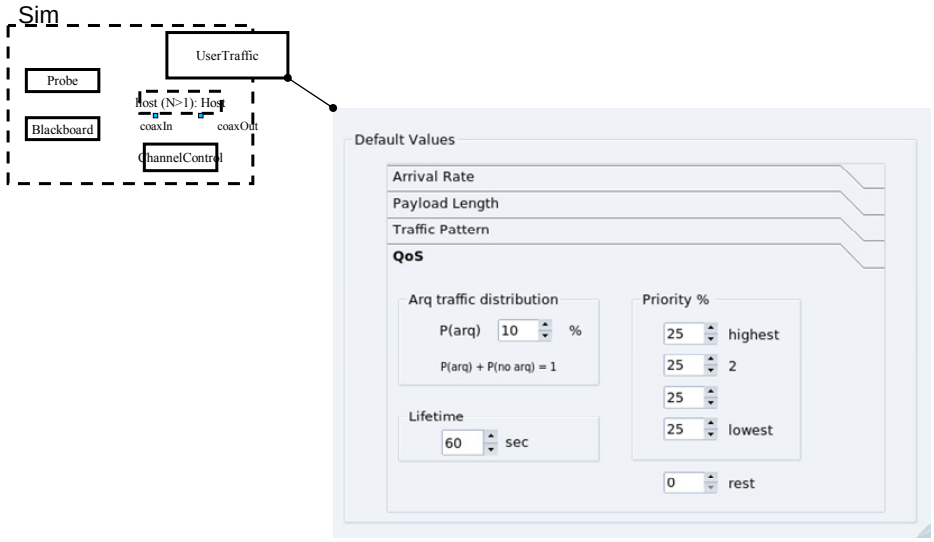


Figure 6.5 The QoS attributes.

The module emulates the users' usage of the application layer services provided by the module *L7_DataProtocol*. The simulator has only one instance of the *UE_UserTrafficDataModule* which serves any number of *Hosts*. *L7_DataProtocol* objects become clients of the *UE_UserTrafficDataModule* when they successfully have passed a registration procedure at run-time. The message sent from this module to the client modules is shown below.

[23] This is a bothersome name and should have been *UserTraffic* only. The name is inherited from another simulator project where a multicast voice model also was implemented. That project has a *L7_VoiceProtocol* module and a *UE_UserTrafficVoiceModule*.

```
message UserData // A message from the user environment
{
fields:
  int destAddr    = Undefined; // range 0...netSize-1
  int srcAddr     = Undefined; // range 0...netSize-1
  int priority    = Undefined; // range 0...3
  bool useArq     = Undefined; // false here disables ARQ within layers 3a,LLC
  double lifetime = Undefined; // [sec]
  int length      = Undefined; // payload length in bytes
}
```

Note that we initialise the attributes to an illegal value which eases debugging.

6.3 The *L7_DataProtocol* Module

The *L7_DataProtocol* module (class *L7_DataProtocol*) is the highest layer protocol. This layer operates in transparent mode, needs no PCI, and the messages received from the *UE_UserTrafficDataModule* module are sent directly down to the *L3_3aLayer* or *L4_Tcp* modules. The *L7_DataProtocol* is the end point for all incoming traffic in the exit nodes, and all incoming messages received are deleted in this module.

```
message L7DataPDU  // (n)-layer peer-to-peer data
{
fields:
      // Layer PCI
      int constPciLength  = 0;      // [bytes]

      // variables for statistics and dbg
      double arrivalTime = StatDbg;
      int serialNumber   = StatDbg;
}

message ApplNetInterface // (n+1)-layer <-> (n)-layer interface data
{
fields:
      int destAddr    = Undefined; // end destination address, range 0.
      int srcAddr     = Undefined; // end source address, range 0...(n-
      int priority    = Undefined; // priority range 0...3
      bool useArq     = Undefined;
      double lifetime = 60.0;      // [sec]
      int sduLength   = Undefined;   // constPciLength + payloadLength
}
```

Figure 6.6 The fields of the application PDU and the interface control information. The data structure shown is compiled by the OMNeT++ meta compiler that produces C++ classes.

6.4 The *L3_3aLayer* Module

The network layer (class *L3_3aLayer*) is implemented as an *OMNeT++* simple module, and the module architecture is shown in the Figure 6.7. The 3a layer protocol is implemented in a dedicated class named *L3_3aPDP* (Packet Data Protocol) which operates on a peer-to-peer basis, that is, the class sees one remote node only. The module has knowledge of its neighbours and creates one *L3_3aPDP* object for each remote peer entity.

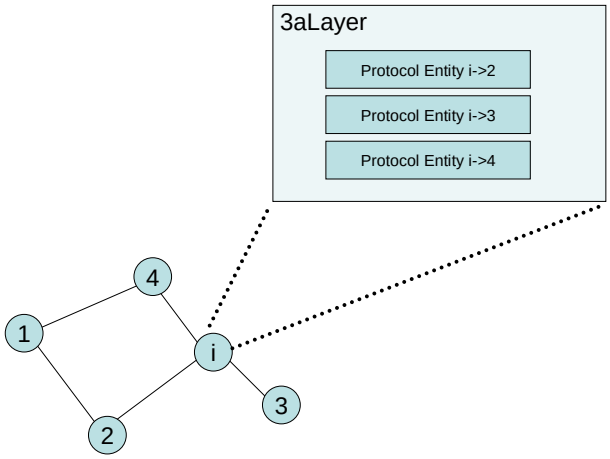


Figure 6.7 The internal data structure of the L3_3aLayer module. Node i creates three separate objects of the class L3_3aPDP; one for each of its three neighbours in the topology shown.

The *L3_3aLayer* module within a host module receives SDUs from the layer above, looks up the next hop address, and forwards the packet to the *L3_3aPDP* object which serves the destination. The *L3_3aPDP* object in charge is determined by the destination address at layer 3. The *L3_3aPDP* object takes over the responsibility for the packet and stores the data packet in its queuing system (class *L3_3aPDP::Buffer*) for unserved data according the packet's priority level. The *L3_3aPDP* works on a point-to-point basis when it forwards packets. The *L3_3aLayer::poller()* allocates execution cycles to the *L3_3aPDP* objects and the *L3_3aLayer* interacts with the LLC layer module on behalf of the *L3_3aPDP* objects.

6.5 The L2_*LlcLayer* Module

The LLC layer enhances the quality of the radio channel by employing a selective-repeat ARQ protocol (transmitter window <= 2). This in conjunction with the precedence and preemption mechanism needed at this layer, makes modelling fairly complex. The LLC layer is implemented as an *OMNeT++* simple module and the module architecture is shown in Figure 6.8.

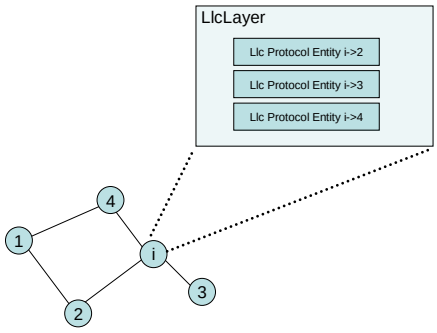


Figure 6.8 The internal data structure of the LLC Layer module. Node i creates three separate objects of the class L2_LlcProtocol; one for each of its three neighbours in the topology shown.

The *L2_LlcLayer* module creates one protocol entity for each of its peer entities (a peer entity is the active element in a host that executes the LLC layer protocol). The class *L2_LlcProtocol* implements the LLC protocol. The *L2_LlcLayer* module within a host module receives LLC SDUs from layer 3, looks up the LLC destination address and routes the packet to the *L2_LlcProtocol* object serving the destination. The *L2_LlcProtocol* object in charge is determined by the LLC destination address received from layer 3. The *L2_LlcProtocol* object takes over the responsibility for the packet and stores the data packet in its queuing system for unserved data according to the priority level. The *L2_LlcProtocol* operates on a point-to-point basis. The *L2_LlcLayer::poller()* allocates execution cycles to the *L2_LlcProtocol* objects and interacts with the MAC layer on behalf of the *L2_LlcProtocol* objects. The local interface flow control between the LLC module and the 3a module is based on a Xon/Xoff message exchange via the control ports.

6.6 The L2_*MacLayer* Module

The MAC layer module (class *L2_MacLayer*) uses no internal buffering and hence introduces no queuing delay. This means that all outgoing packets reside in the LLC module and when served, they are sent to the baseband module immediately. A sequence diagram for successful packet scheduling is shown in Figure 6.9. The LLC module initiates the MAC scheduling by sending a request signal to the input control port of the MAC module telling the priority level to use (t_1), and the MAC starts an internal scheduling timer. If the MAC module receives a busy signal from the radio due to an incoming packet over the air interface, it aborts the scheduling and informs the LLC module about this event (t_2). When the scheduling timer expires, MAC requests data (t_3), and LLC sends the data without any delay to the MAC input data port (t_4). MAC extracts the payload from the SDU, builds a MAC PDU, and then forwards the data with zero delay to the baseband data port. The radio starts serving this packet immediately.

The MAC module is blocked from further service until the radio is able to send more air frames. This happens after the ongoing transmission is completed plus the backoff delay after each transmission.

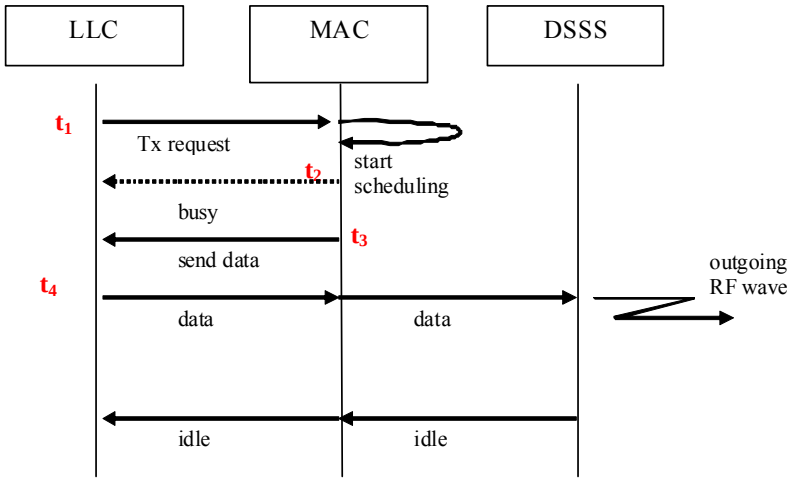


Figure 6.9 Sequence diagram for successful MAC scheduling and sending.

When the radio detects a preamble, the state changes from idle to busy and the radio informs the MAC module immediately by emitting a local busy signal as shown in Figure 6.10. The MAC module informs the LLC module about the state change (t_1), and the LLC cannot request more service before it receives an idle signal. The baseband module sends the incoming packet successfully received, without any delay, to the MAC module (t_2).

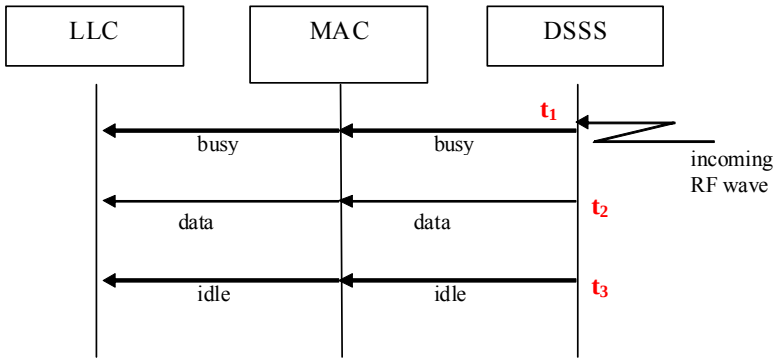


Figure 6.10 Sequence diagram for successful packet reception. Note the local control signals which are forwarded by the MAC module to the LLC module.

6.7 The *L1*_Dsss*Baseband* Module

The *L1_DsssBaseband* module (class *L1_DsssBaseband*) implements the DSSS baseband processor. The air frame reception time sequence and the transmit time sequence are modelled as finite state machines within the class *L1_DsssBaseband,* while processing of events that need signal-to-noise calculations, are handled by the class *L1_RFdecider*.

The channel control module records transmissions, their signal levels and the timing. The *MChannelControl* module makes an explicit copy of the RF waves for each *DsssBaseband* module, and the *DsssBaseband* forwards these RF signals to the *RFdecider*. The *RFdecider* determines the outcome of the *DsssBaseband* correlation/demodulation process from the following information elements:

- The stage of the current captured packet.
- The signal levels of any overlapping transmissions.
- The relative time-delays between any overlapping transmissions.

The most complex part of the *DsssBaseband* is the receive process. With the assistance of Figure 6.11, we outline the normal receive process. Consider a network with one transmission only and a wave arriving at an idle radio at time instance t_1. The RF wave is sufficiently strong, compared to the background noise, to trig a carrier sense (Cas) alarm. When the Cas alarm (preamble detected) occurs at t_2, the *DsssBaseband* emits a "state is busy" signal to the MAC module. The frame demodulation phase starts upon successfully capturing a preamble. From this point in time the radio starts to receive F1 and cannot jump to any overlapping preambles. The *DsssBaseband* sends a "state is idle" signal at t_4 after having entered the sync-search state.

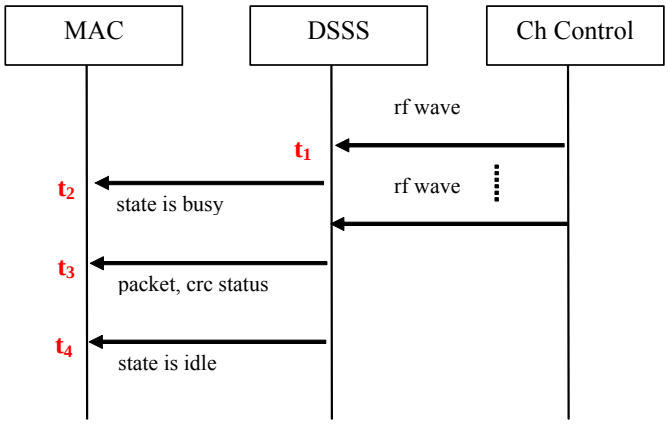


Figure 6.11 Sequence diagram for incoming packet over the air interface

The simulator implements three packet capture models (*L1_RFdecider::decide()*). The purpose of two of these, the **perfect capture** and **zero capture**, are to validate the simulator as well as giving upper and lower bounds during traffic analysis. These capture models do not emulate real radios. A short description of the three capture model follows below.

Normal

This is the normal operating mode described in section 3.4 where preamble detection, F1 detection and the SER are determined by the signal-to-noise level of interfering transmissions. The baseband module keeps track of all the RF waves on the coax input port and performs comprehensive calculations to emulate a real radio.

Perfect capture

When a preamble is detected on the first incoming RF signal with a sufficiently high SNR, the receiver stays locked onto this air frame regardless of the number of overlapping transmissions. F1, CRC16 and LI are always detected correctly. After the complete frame is received, the probability of an error free payload is determined by the user input parameter "P(loss of payload)" in Figure 2.5.

Zero capture

The preamble of the first arriving frame is detected if it has a sufficiently high signal level compared to the background noise. Any overlapping transmission(s) in any stages of a captured air frame always lead to receive failure. After the complete frame is received successfully, the probability of an error free payload is passed to a second process which determine the probability of payload corruption from a single input parameter, the "P(loss of payload)" in Figure 2.5.

We now consider the principles of SNR calculation based on the RF waves. The baseband module starts a Cas timer for every incoming RF wave when it is in the sync-search state (the correlator searches for a preamble). Upon timeout, the corresponding RF wave becomes the analysed signal and all the other RF waves on the channel are treated as interference. The explanation of the processing of the RF waves relies on the terminology "stationary interference" and "dynamic interference". A dynamic interference is caused by a packet transmission which

starts within the vulnerability period of the analysed signal. A stationary interference covers the entire analysis period. When the baseband module analyses packet-B in Figure 6.12, packet-C is treated as a dynamic interference and packet-D as a stationary interference. In a fully connected network, the MAC protocol is able to give a disciplined access to the channel and we seldom experience type D packets. However, as the topology alters and hidden nodes show up, the amount of stationary interference increases.

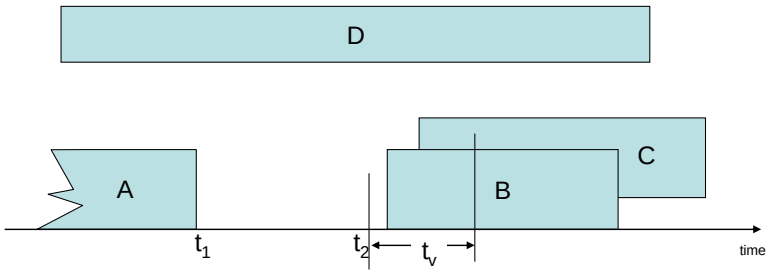


Figure 6.12 A packet transmission ends at t_1 and all busy nodes within the neighbourhood that have a common understanding of an idle channel state start their MAC scheduling.

Each baseband module holds a list of RF waves presented on the coax input port and establishes lists of dynamic interferences and stationary interferences. These lists are processed to form a total noise figure which is the basis for finding the probability of success according to the current signal-to-noise ratio on the radio channel.

The simulator schedules a specific event for each of the captured frame stages; preamble, F1, PLCP-header and payload (see Figure 3.13). Upon processing these events, the simulator calculates the current SNR level that is one of the inputs to a stochastic process which determines further actions. This process is implemented by the methods *L1_DsssBaseband::fsmPreamblePhase*, *L1_DsssBaseband::fsmDemodPhase* and *L1_RFdecider::decide*.

6.8 The M*ChannelControl* Module

A radio in the real world emits a signal that propagates through the air. The counter part of an RF signal in the simulator is modelled as the data structure shown in Figure 6.13. Every time a radio emits a wave, the baseband module creates an instance of the class *RFwave*, inserts the transmitting power level measured at its coax output port and sends the wave to the channel control. The basic idea is that the channel control shall calculate the signal power level at the coax input port for all other radios in the network and forward an explicit copy of the *RFwave* to each baseband module.

```
message RFwave
{
fields:
double txPowerDbm;  // Radiated power in dBm with which this packet is transmitted.
double rxPowerDbm;  // Power in dBm at the receiver antenna input
double rxPowerW;    // rxPowerDbm converted to W
double duration;    // Time it takes to transmit the packet, in seconds!
double startedAt;   // The time instance the wave reached the receiver antenna.
int srcNode;        // The host identifier (>= 0) that sends this packet.
};
```

Figure 6.13 The attributes of the RF wave. The transmitter inserts the txPowerDbm while the channel control calculates the rxPower.

7 Tips and Tricks

This chapter offers a number of hints about the practical usage of *oTWLAN*.

7.1 Sanity Checks of the Input and Output Data

Sanity checks of the output data are to inspect a number of output files with the objective to find errors, either in the software or the input data files. One aid we have is the counters explained earlier in this document (cf. the *OMNeT++* term *scalars*). A counter is a simple variable within the source code that counts something, e.g., a variable is incremented whenever a packet is lost, or a broadcast is received. The simulator has implemented many counters and they are written into the file *counters_r1*. The file name is tagged by the session run number it belongs to - "*_r1*" in the example name emphasis that the file is an output from run number 1. Consider the three-node chain in Figure 7.1.

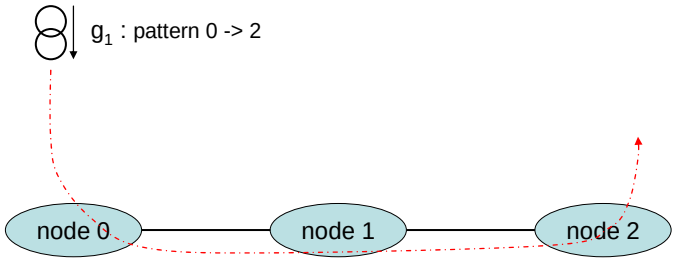


Figure 7.1 A chain of three nodes with one active traffic generator. The generator sends packets from end source 0 to end destination 2, and does not request use of ARQ.

Here is the output from the counters within the LLC layer:

```
sim.host[0].nic.llc::L2_LlcLayer:noOfSDUsReceived = 6000
sim.host[0].nic.llc::L2_LlcLayer:noOfDataPDUsReceived = 0
sim.host[0].nic.llc::L2_LlcLayer:noOfBroadcastsReceived = 6000
sim.host[0].nic.llc::L2_LlcLayer:noOfAcksReceived = 0

sim.host[1].nic.llc::L2_LlcLayer:noOfSDUsReceived = 6000
sim.host[1].nic.llc::L2_LlcLayer:noOfDataPDUsReceived = 6000
sim.host[1].nic.llc::L2_LlcLayer:noOfBroadcastsReceived = 0
sim.host[1].nic.llc::L2_LlcLayer:noOfAcksReceived = 0

sim.host[2].nic.llc::L2_LlcLayer:noOfSDUsReceived = 0
sim.host[2].nic.llc::L2_LlcLayer:noOfDataPDUsReceived = 6000
sim.host[2].nic.llc::L2_LlcLayer:noOfBroadcastsReceived = 0
sim.host[2].nic.llc::L2_LlcLayer:noOfAcksReceived = 0
```

This network is configured to have radio channels with an excellent SNR level and the offered traffic is far below the throughput capacity. Thus packet loss shall not occur. The counters confirm the following important facts about the network:

- node 2 shall only receive PDUs and no SDUs since its 3a layer shall not send data
- node 0 shall only receive SDUs and no PDUs since the only traffic generator we have is in node 0
- node 1 shall receive both SDUs and PDUs, and the numbers shall be identical.
- none of the nodes shall receive acknowledgement packets

The latter bullet is obvious because ARQ is disabled. However, a frequent cause of errors is incorrect setting of the traffic generators and additional checks against the counters often discover errors in the simulator's input data. Additional aid you can use before starting the simulator's kernel thread is the status widgets in Figure 7.2.

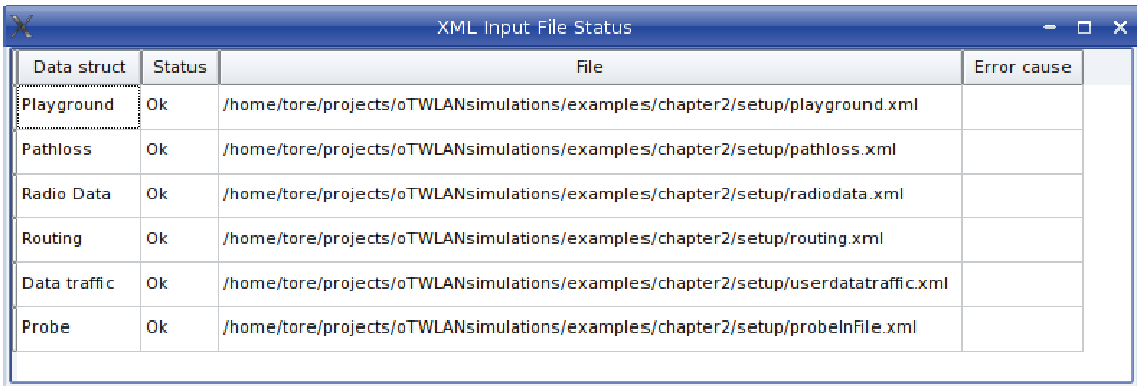


Figure 7.2 The XML Input File Status widget is activated by the meny "View->XML File Status".

7.2 How to simulate without the GUI part

We have so far described simulation runs from the GUI front-end where a single network scene is taken from a single setup directory. Often we have many different network scenes to simulate, and typically we want to start a bunch of simulations at Friday to have the results ready at Monday morning. Batch simulation without use of the GUI is possible when each network scene is specified in separate setup-directories. If you run the simulator as a console application (class *GUI_SimWithoutGui*) using the *runSet* and *simDir* arguments below then the program starts without the GUI:

yourBinName **-runSet** *yourRunSet* **-simDir** *fullPathToSimulationHomeDirectory*

The *yourRunSet* confirms to the OMNeT++ syntax (1-5, 1,2,4, etc).

Assume you want to estimate throughput as function of the offered traffic at the three different power levels: {1W, 2W, 3W}. Then use the GUI front-end to create a home directory named

power1W, specify the offered traffic (run set 1 to 7), specify the other data and set the transmitting power to 1W. Copy this directory to two new directories named *power2W* and *power3W*, and set the transmitting power to 2W and 3W, respectively. Then you open a console/xterm window and writes[24]:

yourBinName - **runSet** *1-7* **-simDir** */home/tore/simulations/power1W*
yourBinName - **runSet** *1-7* **-simDir** */home/tore/simulations/power2W*
yourBinName - **runSet** *1-7* **-simDir** */home/tore/simulations/power3W*

The outputs will be placed in *power1W/output*, *power2W/output*, *power3W/output*, respectively.

7.3 How to remove the GUI software

This situation occurs if you want to remove the GUI software (remember that simulation without the GUI is possible). Figure 7.3 shows the software dependency graph and we suggest that you start to remove the cpp-files prefixed by GUI. Use the compiler to determine the next steps.

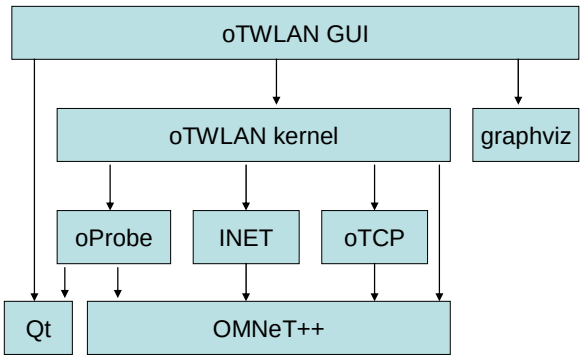


Figure 7.3 Software dependency graph. oTCP is an optional module.

7.4 How to remove the kernel part

This situation occurs if you want to reuse the source code that implements the input data editors. We suggest you start out with the class *GUI_MainWindow* as basis for your software. *KDevelop* users shall establish a new project of the type <C++><QMake project><Qt4 Application> and create an object for the *GUI_MainWindow* in the *main.cpp* as we have in our source code. Then use the compiler to resolve missing classes and modify the code to prevent dependency of the class *oProbe::OMNET_Thread*.

[24] Hint: Use a text editor to do a copy/paste into the console window, or write a script.

8 The Software Architecture

The purpose of this chapter is to analyse the software environment for the *oTWLAN* project and end up with a naming convention for C++ class names. The analysis model below expresses that we are bounded by three external interfaces. A *UI*[25]-package provides widgets to the human user by which he can edit input parameters, perform execution control and handle output data in different formats. A *MySimulator*-package contains the functions required to implement an *oTWLAN* simulator based on the *OMNeT ++* software components. The simulator needs functions to collect data and execute data analysis in real-time. These functions are provided by the *oProbe*-package. *oProbe* is an external software package described in [2].

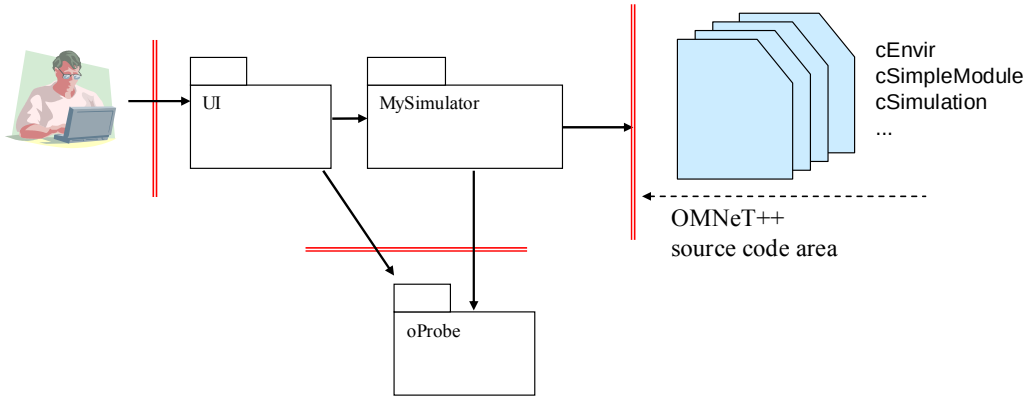


Figure 8.1 The analysis model for the oTWLAN project. The three most important interfaces to the external environment are shown as red lines. The solid arrows should be read as "depends on" (e.g., the UI-package depends on the oProbe-package).

The *MySimulator*-package implements the modules defined in the previous chapter by using the *OMNeT ++* API. Data shall not be sent directly from the *MySimulator*-package to the *OMNeT++*, but shall be stored in files and read via the *OMNeT++* kernel when the simulator starts (cf. *OMNeT++::initialize()*). The same principle shall apply to output data (cf. *OMNeT++::finish()*).

Building a simulator is to implement *OMNeT++* modules as C++ classes and to build a front-end based on the Qt class library [7], see Figure 8.2. The GUI widgets are based on the Qt GUI module and when we program classes that shall operate "near the user" (front-end classes) the functionality of the Qt API is used as much as possible. In the opposite direction we have the *OMNeT++* section where the simple modules are implemented.

The input data is configured via GUI widgets and saved into XML based files. When the user activates "start simulation" from GUI kernel launcher (class *GUI_OmnetStartEditor*), *oTWLAN* builds the *omnetpp.ini* file automatically (class *UTL_OmnetIni*) and starts the *OMNeT++* kernel as a thread (class *oProbe::OMNET_thread*). The modules take their input data from the XML files.

[25] User Interface

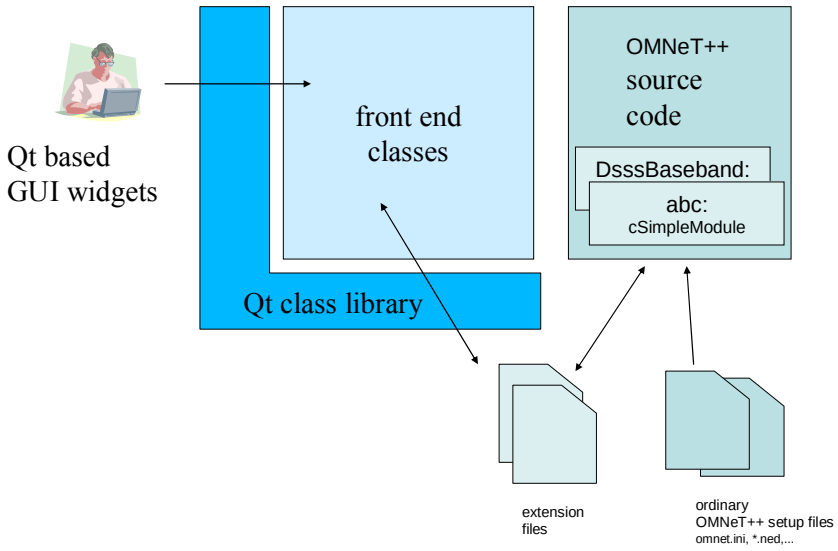


Figure 8.2 The software architecture.

The analysis model in Figure 8.1 is expanded to the design package diagram shown in Figure 8.3, and the packages outside the red dotted rectangle are external packages. The analysis package *UI* (User Interface) is expanded to the packages *GUI* (Graphical User Interface) and *GUIA* (GUI Automatic). This project uses the Qt4 Designer for widget production; the C++ classes produced by this tool belong to the *GUIA*-package. The *GUI*-package uses the *GUIA*-package to build widget for interaction with the user. The XML-package is based on the *QtXml*-module and the classes using this module belong to this package. Classes implementing the UE-module belong to the UE-package. *oTWLAN* has a separate *OMNeT++* module for each OSI layer and we introduce a software package for grouping the software that deals with the protocol stack. To concretise, the class *L3_3aPDP* implements functions belonging to the OSI layer 3 (L3) and the "3aPDP" indicates that this is the 3a layer Packet Data Protocol (PDP).

Every software project has a set of class which has general usage or is difficult to put in a specific category. *oTWLAN* is no exception and we specify a UTL[26]-package as a container for these.

[26] utility package

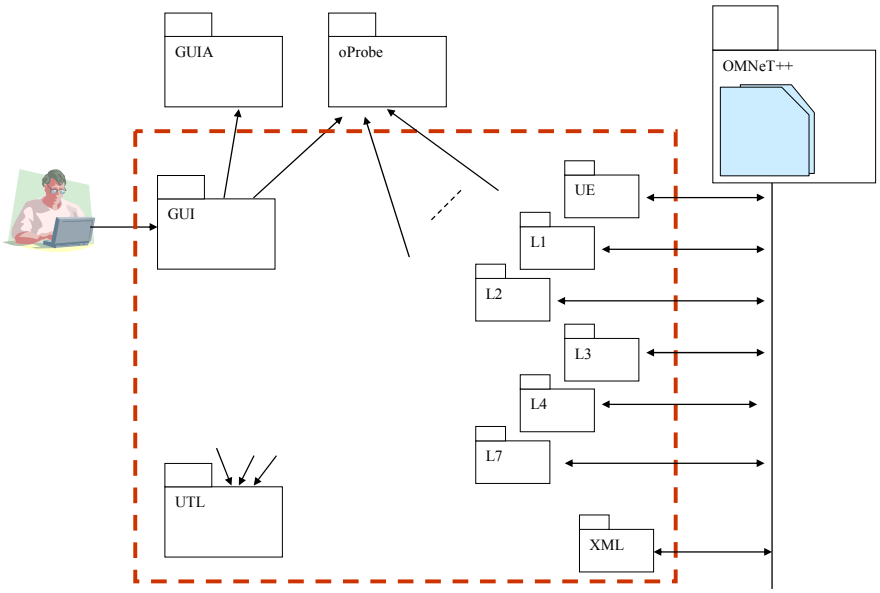


Figure 8.3 Design packages and their relationship.

The *UE*-package contains the classes for modelling the usage of the application layer services, while the *oTWLAN* protocol layer modules specified in the design chapter belong to the *Lx*-package, where *x* denotes the number of a layer. The *XML*-package contains the classes that handle transformation of data to and from XML. The package *UTL* (Utility) contains classes of general usage, and is used by many internal packages.

A problem that often occurs when combining a number of open source projects is name clashes in C++ programs. A namespace is a mechanism for reducing the risk of this problem. Therefore, we put the namespace name OTWLAN around all declarations in the header files.

The file and class name convention used for the project is to tag the names with a prefix to identify the design/implementation package they belong to. In our project, we do not find it practical to differentiate between a design package diagram and an implementation package diagram. When you see ZZZZ_zzzz, you know that this class belongs to the ZZZZ-package defined in Figure 8.3, while zzzz expresses something about its functionality. For example, *L2_LlcLayer* belongs to the L2-package (software that implements OSI layer 2 functions) and implements the LLC layer as a *cSimpleModule*, while the *L2_LlcProtocol* is a class that implements the LLC layer protocol.

9 Validation and Parameter Optimization

The purpose of this chapter is to evaluate the correctness of the simulator and find protocol parameter values which optimize the network throughout. We must have an analytical performance model as a reference to achieve the task. By defining a simplified operating scenario, we can use the MAC throughput model outlined in [5, chapter 5]. The following parameters remain constant in this chapter:

Network topology - "all-hearing-all" (AHA)
The radio link pathloss model is set to 10 dB fixed loss and the transmitting power to 1W. These values give high quality radio links and any observed degradations must be caused by the intra network layer protocols.

Layer 7 Offered traffic
Arrival distribution: Poisson with variable rate
Layer 7 payload size: fixed 400 bytes
Pattern: Uniformly distributed over all destination addresses
Lifetime: 60 sec

The simulator supports three capture models and this chapter uses the normal capture model if not otherwise noted. Under low collision rates this model shall behave as the perfect capture model due to the excellent radio conditions given; the signal level at the receiver side is 20 dBm. The transmission time for an air frame carrying a layer 7 payload of size LI_7 bytes is given by:

Equation Chapter 9 Section 9

$$t_{dt} = t_v + t_{F1} + 8 \cdot (20 + LI_7) / f_{payload} \quad (9.1)$$

The transmission time for an LLC acknowledgement is given by:

$$t_{ack} = t_v + t_{F1} + 8 \cdot 11 / f_{payload} \quad (9.2)$$

The PCI sizes are stated in Table 9.1 while the other parameters are given by Table 3.1. Given fixed sized L7 payload, the perfect capture model and low collision rates, the average channel busy period is approximately identical to the air frame size[27]. An analytical expression for the throughput capacity for ARQ traffic is then:

$$\lambda_{p,\lim} = p_{net} \cdot LI_7 / (E[C_I] + t_{dt} + t_{ack} \cdot p_{net}(n + 1 - p_{net}) / n) \text{ [bytes/s]} \quad (9.3)$$

This expression becomes inaccurate as the bit-error rate increases since the nodes turn to backoff after detecting bit errors, but comply reasonably well for the perfect capture model where the first

[27] The channel busy period range is $[t_{dt}, t_{dt} + t_v]$ as long as the nodes have the correct understanding of the channel state.

packet succeeds in case of collisions. Nodes also turn to backoff after a transmission but this effect becomes small in large networks.

Layer	DT PCI [bytes]	LLC ACK PCI [bytes]	ACK at 3a layer
L7	0	-	0
L3a	6	-	4
LLC	4	1	4
MAC	6		
PHY	4 (F1 excluded)		
Total:	20	11	18

Table 9.1 Protocol Control Information (PCI) sizes.

The nominator is the time delay to deliver the packet and to receive the acknowledgement. p_{net} is the probability of successful packet reception and is given by [5]:

$$p_{net} = \frac{n}{n-1}\left[1 - \frac{1}{n} - \frac{1}{b_p} + \frac{1}{n}\left(\frac{1}{b_p}\right)^n\right] \tag{9.4}$$

We have mentioned earlier that the optimum b_p-value depends on the traffic conditions. To illustrate this, Figure 9.1 plots the normalised throughput versus $b_{p=3}$ under different layer 7 payload lengths. Normalised throughput is the throughput from equation (9.3) divided by the air frame payload transmission rate in bytes per second. The figure illustrates clearly that it is better to select a large b_p -value than a small value. The figure gives also an impression of the increased cost of sending short packets relative to long packets in a fully-connected network.

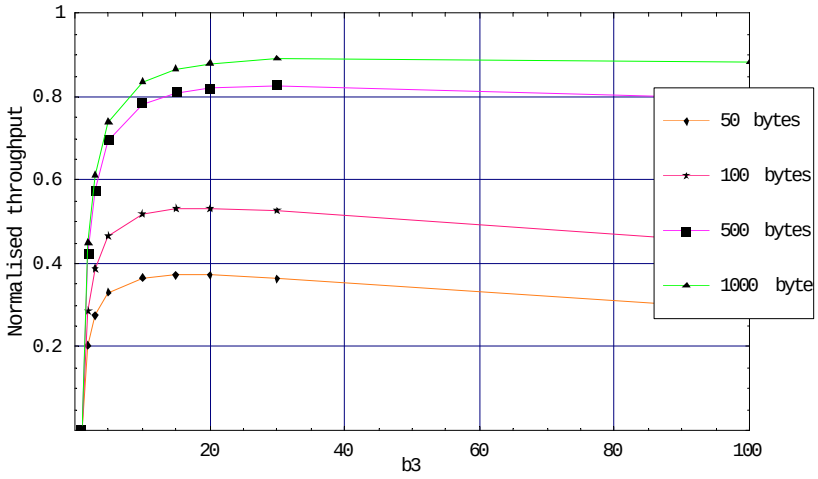


Figure 9.1 Normalised throughput as function of b_3 in a network with 25 active nodes. The legend expresses the L7 payload size.

Some readers may find the ordering of the forthcoming sections discursive since they are not strictly ordered according to validation/optimalisation. Sections 9.1, 9.4 and 9.6 deal with validation and sections 9.2, 9.3 and 9.5 consider optimalisation.

9.1 AHAn2

The purpose of this section is to validate the correctness of the simulator's time base. We use a two-node network with simplex deterministic traffic since this give an easy scenario to analyse. Node 0 transmits to node 1 using a deterministic packet arrival distribution with $\Lambda = 1.0$ packets/s and the fixed payload size 100 bytes. The channel busy period measured by node 1 is given by $t_{dt} - t_{on} - t_{cas}$ msec since node 1 detects a busy channel after having detected the preamble, while the channel idle period becomes $1/\Lambda - t_{dt} + t_{on} + t_{cas}$. We use $(a_p, b_p) = (3, 20)$, one priority level only and ARQ is disabled. The power level is set sufficiently high to have an error-free radio channel.

The system operates at a low load level and only the MAC service time contributes to the end-to-end delay, given by $t_v(a_p + b_p/2) + t_{dt}$, which equals to 25.88 msec at 100kbps. The minimum end-to-end delay is given by $a_p t_v + t_{dt} = 14.48$ msec and the maximum is $(a_p + b_p)t_v + t_{dt} = 37.28$ msec.

Table 9.2 compares simulated results and theoretical results, and the table show excellent conformity between theoretical and simulated results.

Estimators		Radio version		
		100 kbps	1 Mbps	10 Mbps
Channel idle period [sec]	simulated	$0.99008 \pm 1.8 \cdot 10^{-5}$	$0.999008 \pm 1.8 \cdot 10^{-6}$	$0.999901 \pm 1.8 \cdot 10^{-7}$
	analytical	0.99008	0.999008	0.999901
End-to-end delay [msec]	simulated	25.7484±0.18	2.57484±0.018	$0.257484 \pm 1.8 \cdot 10^{-3}$
	analytical	25.88	2.588	0.2588
End-to-end delay (min,max) [msec]	simulated	(14.48, 37.28)	(1.448, 3.728)	(0.1448, 0.3728)
	analytical	(14.48, 37.28)	(1.448, 3.728)	(0.1448, 0.3728)

Table 9.2 Comparison table between simulated and analytical results with ARQ disabled. Estimations of first order moments are presented as 95% confidence intervals.

9.2 Optimising b_p

The aim of this section is to find a value for the priority delay b_p given the single priority level P3. The b_p parameter is the most important parameter for the efficiency of the MAC protocol since it regulates the probability of collision (equation 3.1) and the channel idle period (equation 3.2). These equations are only valid for the heavy-load case where all nodes have at least one packet in the buffers. It is not possible to find a single value that gives maximum throughput under all conditions because throughput depends on many parameters; the number of nodes in the network, the network load level and the packet size distribution, to give a few examples.

Figure 9.2 plots the impact of the priority delay factor when using the single priority level P3. The throughput performance under the b_3-set {5,10,20} suffers from a high collision rate. This

conclusion can be drawn by the shape of the plot; the throughput drops from a maximum. The b_3-set {100,200,500} gives neither maximum throughput; the collision rate is low but the average idle channel period is too long. The optimum b_3-value under this traffic condition is 50.

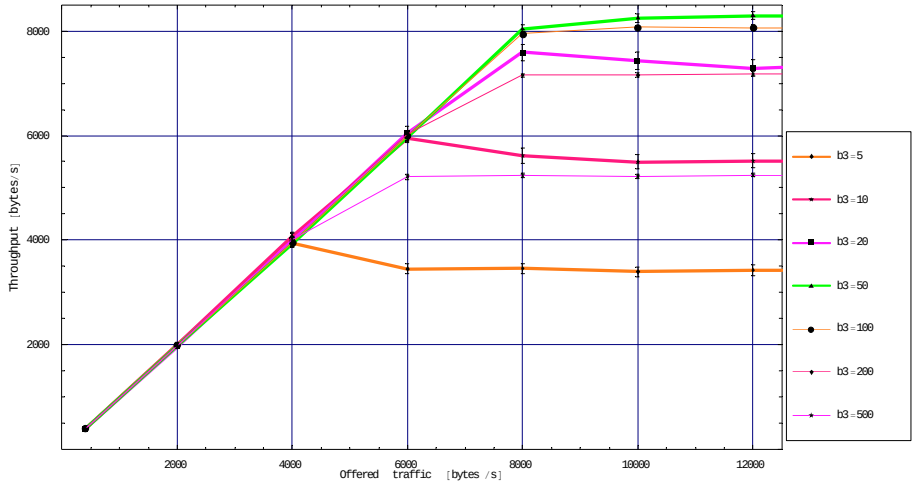


Figure 9.2 Throughput (90% confidence intervals) as function of the offered traffic. Only priority P3 traffic is used. Radio version: 100kbp (simDec11a).

Figure 9.3 substantiates the statement of performance degradation due to a high collision rate for $b_3 = 5$ since the number of retransmissions by the ARQ protocol within the LLC layer is very high compared to the $b_3 = 500$ case.

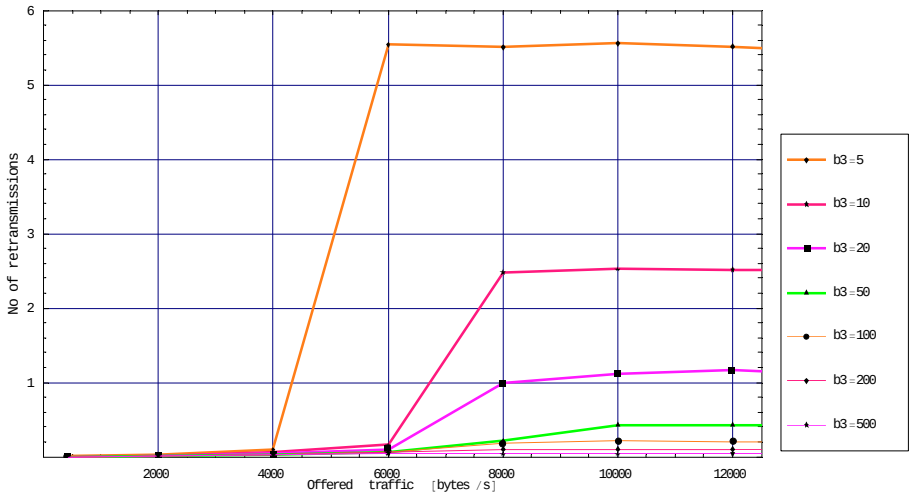


Figure 9.3 The course of the number of retransmission with increasing load level for radio version 100kbps. Estimation is done without confidence control (simDec11a).

How can we be certain of that our interpretation of the situation is correct and is not caused by a software bug? Here is where the counters become a valuable tool. A printout of some of the counters from the $b_3 = 5$ simulations at maximum load is:

```
sim.host[0].nic.baseband::L1_DsssBaseband:noOfCasAlarms = 39947
sim.host[0].nic.baseband::L1_DsssBaseband:noOfRfWavesRxed = 78499
sim.host[0].nic.baseband::L1_DsssBaseband:noOfF1Errors = 2
sim.host[0].nic.baseband::L1_DsssBaseband:noOfPlcpErrors = 5576
sim.host[0].nic.baseband::L1_DsssBaseband:noOfCrc32Errors = 4807
```

The same counter set for b_3=500:

```
sim.host[0].nic.baseband::L1_DsssBaseband:noOfCasAlarms = 20272
sim.host[0].nic.baseband::L1_DsssBaseband:noOfRfWavesRxed = 20489
sim.host[0].nic.baseband::L1_DsssBaseband:noOfF1Errors = 0
sim.host[0].nic.baseband::L1_DsssBaseband:noOfPlcpErrors = 2
sim.host[0].nic.baseband::L1_DsssBaseband:noOfCrc32Errors = 70
```

The data confirms a considerably lower air frame corruption rate for the $b_3 = 500$ case.

Figure 9.4 and Figure 9.5 show the simulated results for the two other radio versions. Maximum throughput is reached for $b_3 = 50$ in both cases. This is identical with the 100kbps case and is expected since all three radio versions use the same preamble/SER probability distributions. The difference is only the chip rate.

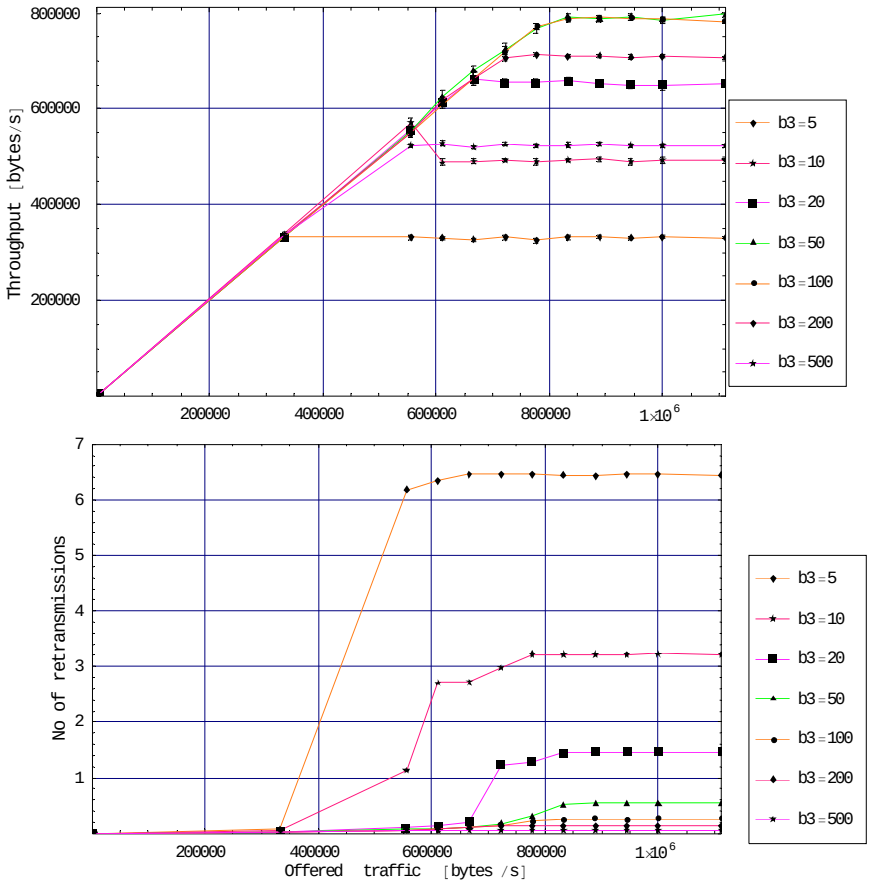


Figure 9.4 Throughput as 90% confidence intervals (top picture) for the 10Mbps radio version. The picture below shows the course of the number of retransmission (no confidence control) (simDec11b).

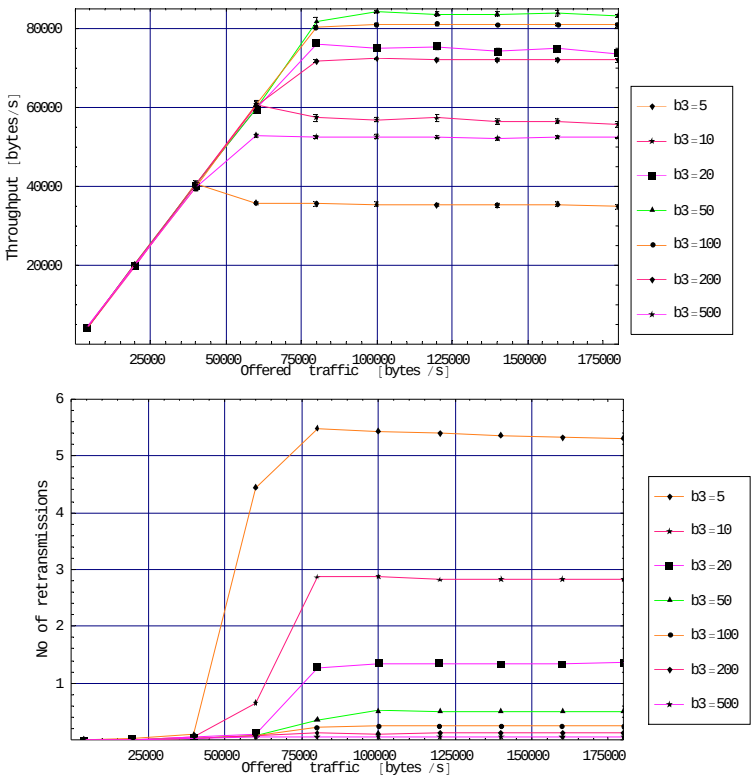


Figure 9.5 Throughput as 90% confidence intervals (top picture) for the 1Mbps radio version. The picture below shows the course of the number of retransmission (no confidence control) (simDec11c).

9.3 Selecting (a_p,b_p)-values

There exists no single b_p-set that maximises the network throughput under all traffic conditions. When b_p shall be considered, we must have some guidelines on the offered traffic priority distribution. If not, the system may suffer from a high collision rate.

Yet we have not looked at the priority delay factor a_p. a_p does not affect the collision rate directly as the b_p parameter. Its function is to give a fixed MAC level access delay such that a high priority packet in node A is granted access to the radio channel before a lower priority packet in node B, given that both nodes start the MAC scheduling process at the same time instance. We select the a_p-set $a_3 = 3,\ a_2 = 4,\ a_1 = 5,\ a_0 = 6$. Hence the lowest priority packets get a fixed delay of $6t_v$ while the highest priority packets get the fixed delay $3t_v$. Any MAC entity scheduling a low priority packet aborts its scheduling when it detects a busy channel t_v-second after a higher priority packet is sent on the air.

Let Λ denote the offered network traffic and Λ_p the offered network traffic at priority level p. In this document, we use the offered traffic priority distribution {P3,P2,P1,P0}={0.1,0.2,0.3,0.4} such that

$$\Lambda_{P3} = 0.1 \cdot \Lambda,\ \Lambda_{P2} = 0.2 \cdot \Lambda,\ \Lambda_{P1} = 0.3 \cdot \Lambda,\ \Lambda_{P0} = 0.4 \cdot \Lambda$$

Section 9.2 "Optimising b_p" found $b_3 = 50$ to be a good choice for the traffic conditions given. Here in this section, we assume a nearly identical traffic condition with the exception of multiple priority levels using the priority distribution above. If we fail to select a non-optimum value, the best what can happen is to select a larger value to achieve a lower collision probability. This is especially beneficial for multihop networks where the MAC protocol is less efficient due to hidden nodes. If we select the b_p-set $\{b_3 = 20,\ b_2 = 30,\ b_1 = 50,\ b_0 = 100\}$ then section 9.2 concluded that we experience non-optimal parameter values when the offered traffic becomes so high that only priority P3 packets are served. However, this is not a normal operating mode and we consider this b_p-set further by conduction some simulation experiments.

Figure 9.6 to Figure 9.8 present the simulated throughput and shows that the MLPP function works perfectly. However, there is a question about the overall capacity loss compared to a network without priority handling.

Section 9.2 optimised b_p for the single priority level P3 and found the network throughput capacity in Table 9.3. The rightmost column in the table presents the cumulative throughput for P0...P3 measured in this section. By comparing the two columns, we get an indication of the cost of implementing the MLPP in terms of throughput capacity loss, which is approximately 3%.

Radio version	Section 9.2	This section
100 kbps	8294 ± 83	8184 ± 90
1 Mbps	83821 ± 562	81520 ± 534
10 Mbps	797368 ± 4575	769102 ± 4524

Table 9.3 Network throughput capacity [bytes/s].

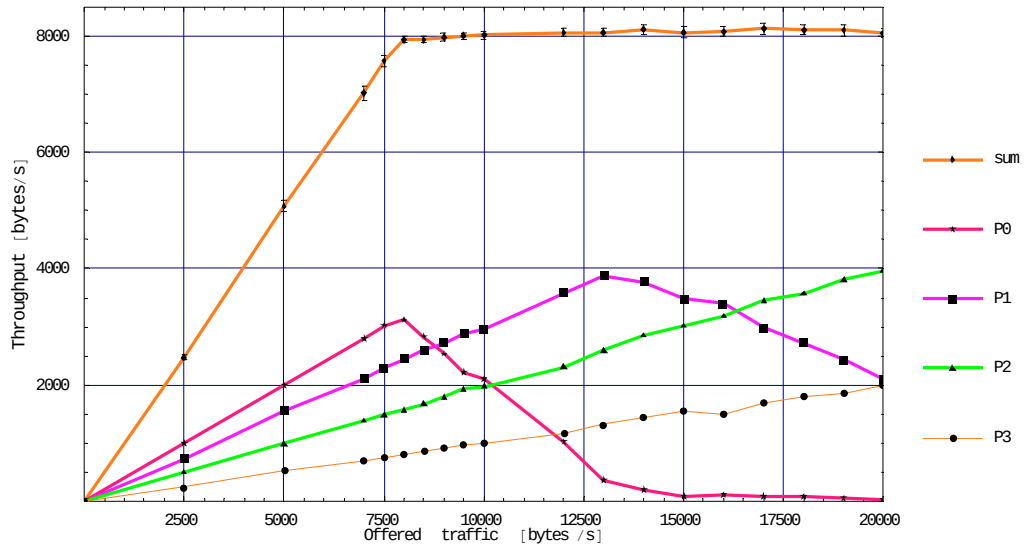


Figure 9.6 Throughput versus offered traffic. No confidence control is used for the plots showing throughput per priority. Radio version: 100kbps (simNov26a).

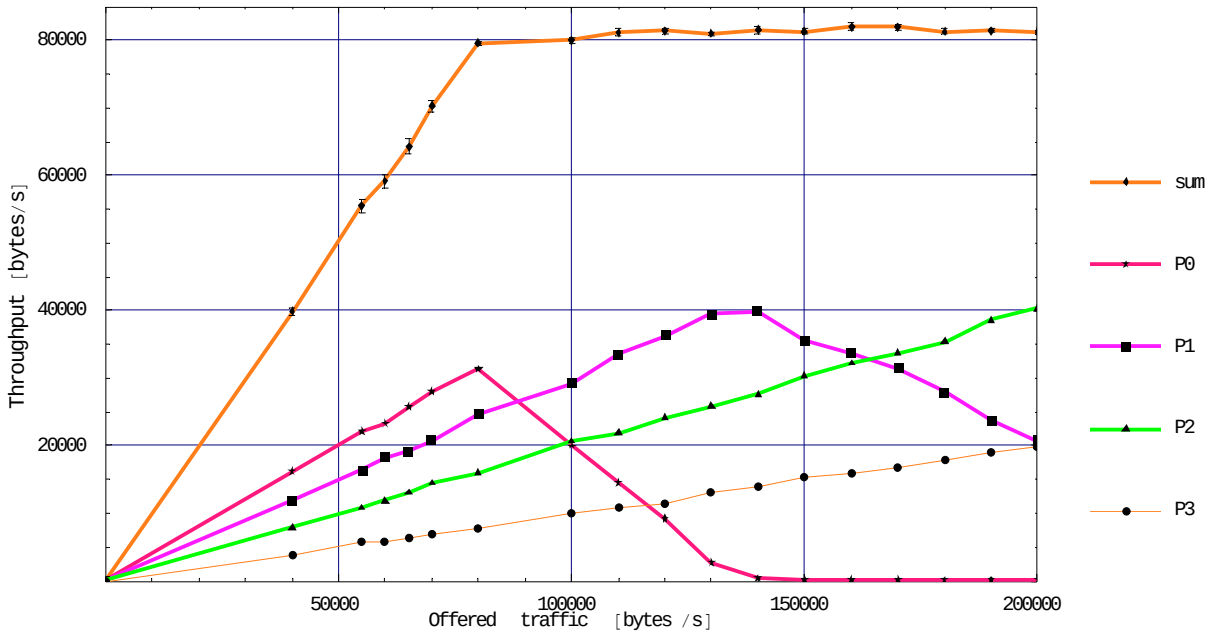


Figure 9.7 Throughput versus offered traffic. No confidence control is used for the plots showing throughput per priority. Radio version: 1Mbps (simNov26c).

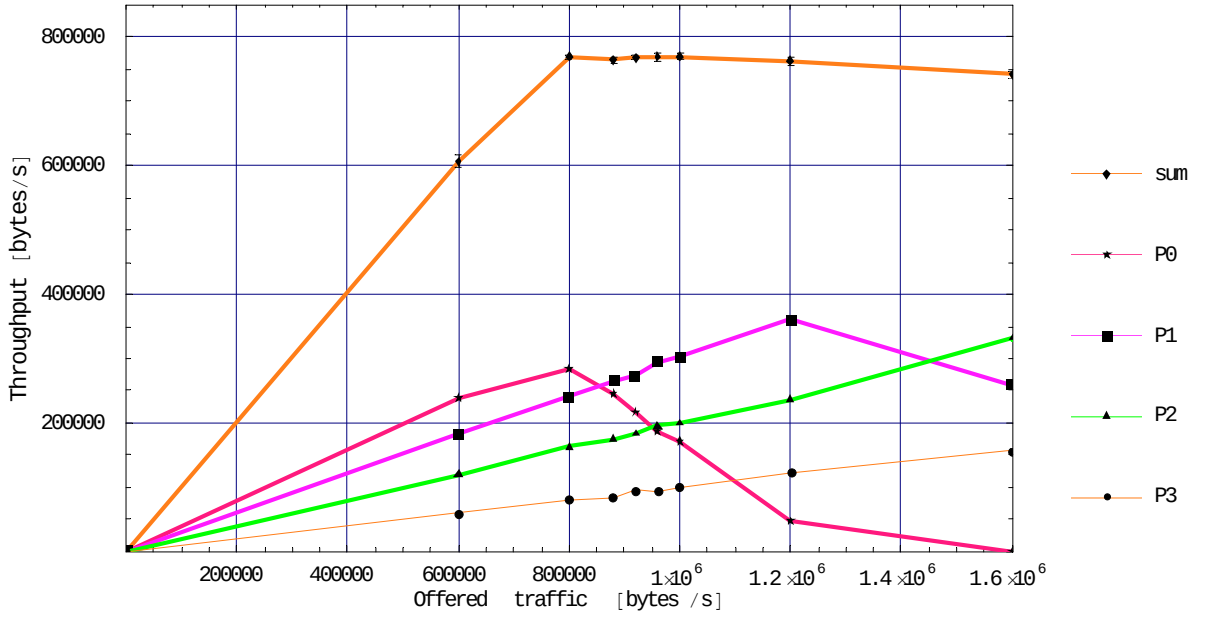


Figure 9.8 Throughput versus offered traffic. No confidence control is used for the plots showing throughput per priority. Radio version: 10Mbps (simNov26b).

9.4 Capacity per priority

The purpose of this section is to make further investigation of the a_p-set and the b_p-set proposed in the previous section. The scenario considered is still a fully-connected network of 16 nodes having the same excellent radio condition. The offered traffic distribution remains identical unless otherwise stated.

The main performance metric in this section is the throughput capacity **per priority** under the condition of using **one** single priority level at a time. The priority levels use different b_p -values and hence the packets experience different collision rates at the network saturation point. Besides the main goal, our ambition here is also to validate the correctness of the simulator.

It is important to have established a set of invariants before any experiment starts. For this section, the following invariants apply:

- I_1: The zero-capture model shall give the lowest throughput performance compared to the other two models.
- I_2: The performance under the perfect capture model shall outperform all other capture models.
- I_3: The performance under the normal capture model shall always be embraced by the throughput for the two other models where the zero capture case represents the lower limit.

I_1 follows from the fact that packet reception never succeed in case of collision under the zero-capture model. I_3 follows from I_1 and that the normal capture model accepts one overlapping transmission[28] while the perfect capture model accepts any number of overlapping transmissions.

The implementation of the perfect and the zero capture models are less complex than the normal capture model and thus are less prone to software bugs. The perfect capture model also has the benefit that it confirms reasonably well to the analytical throughput model.

Figure 9.9 shows simulated throughput versus offered traffic for priority levels P3 and P0. Under low traffic loads the system experiences few collisions and the performance of the three capture models shall be identical. This is confirmed by the plots which show a linear progress up to the offered traffic rate 40000 bytes/s (100 packets/s).

Based on the theoretical throughput model from equation (9.3), we have a means to calculate the performance for the perfect capture case. The theoretical probability of collision at the saturation point for priority P3-traffic is 0.56 compared to 0.15 for priority P0-traffic. Despite the higher collision rate for the P3-case, it outperforms the P0-case since the P3-case has a shorter $E[C_I]$

[28] This is determined by the modulation and the coding (class *RadioSER*).

and the first packet of two overlapping packets succeed. The green horizontal lines in the figure represent the theoretical throughput capacity.

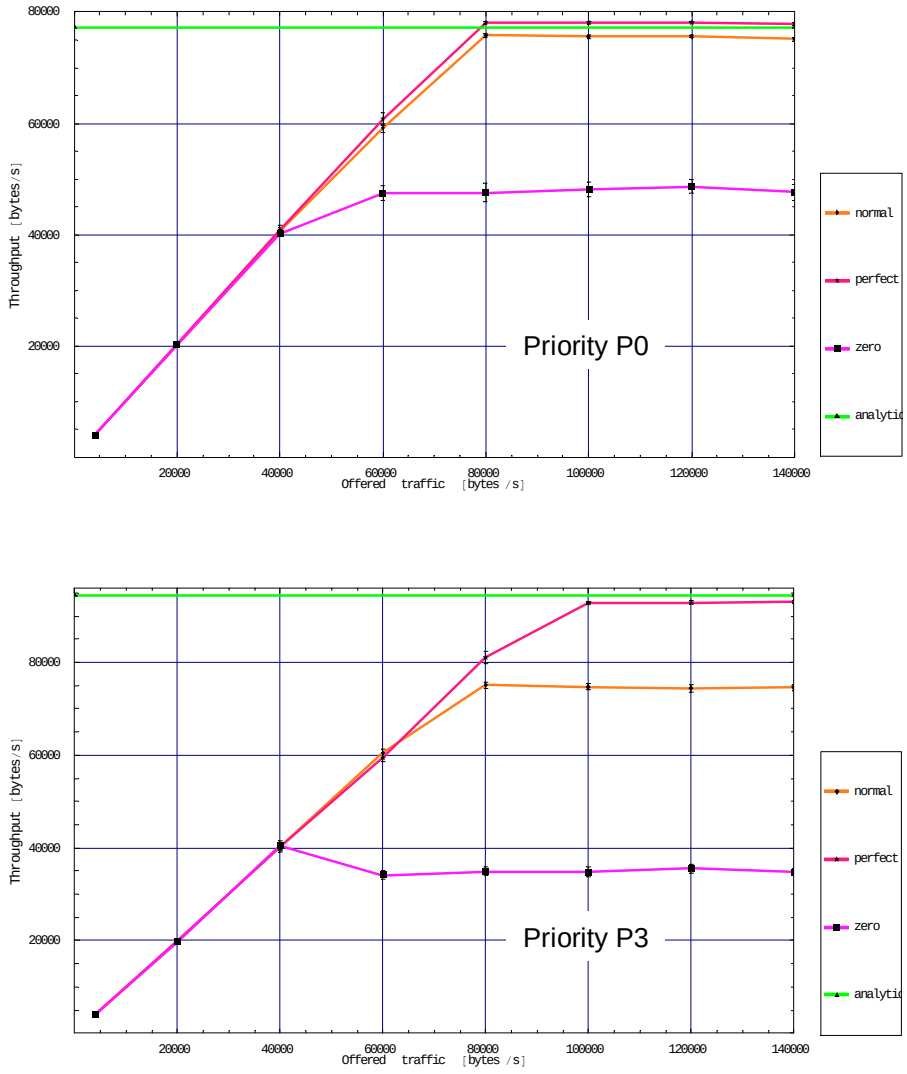


Figure 9.9 Network throughput under different capture models. The offered traffic uses the single priority level marked on the plots. (simJan5a).

The throughput capacity under the zero-capture model degrades faster with increasing collision rates since packet reception always fails in case of collision. Thus the system performance can be improved by increasing b_3; the collision rate decreases but $E[C_I]$ enlarges.

Table 9.4 and Table 9.5 present a more detailed comparison between simulated and theoretical results. We have no theoretical expression for the normal capture model but know that its throughput capacity shall be lower than the perfect capture model. The perfect capture model is compared with the analytical model and the largest deviation we observe is 2.7%.

Network capacity [bytes/sec]		Capture model	
		Normal	Perfect
P3 (highest)	simulated	75135±418	<91796, 92207>
	analytical		90065 (-1.9%)
P2	simulated	78855±510	<88986, 89235>
	analytical		87889 (-1.2%)
P1	simulated	79527±444	<84791, 85067>
	analytical		84357 (-0.5%)
P0 (lowest)	simulated	74562±604	<77596, 77800>
	analytical		77713

Table 9.4 Comparison of simulated and theoretical results of network capacity per priority. The normal capture model is included to illustrate the performance of a "real" radio. Radio version: 1Mbps (simJan9a).

Network capacity [bytes/sec]		Radio version	
		100k	10Mbps
P3 (highest)	simulated	<9146, 9184>	<925907, 931372>
	analytical	9006 (-1.5%)	900649 (-2.7%)
P2	simulated	<8842, 8903>	<895841, 900158>
	analytical	8789 (-0.6%)	878893 (-1.9%)
P1	simulated	<8439, 8503>	<853016, 858023>
	analytical	8436	843568 (-1.1%)
P0 (lowest)	simulated	<7735, 7794>	<777140, 782539>
	analytical	7771	777129

Table 9.5 Comparison of simulated results and theoretical results under the perfect capture model.

The simulator operates with a lower probability of collision since the nodes are forced into a backoff state after transmitting a data packet and after detecting bit-errors in incoming packets. The perfect capture model shall not experience bit-errors with our input data and receipt failure shall be caused only by preamble detection failure. The counter output confirms this and the following section is reprint from a randomly selected host:

```
sim.host[10].nic.baseband::L1_DsssBaseband:noOfCasAlarms = 59061
sim.host[10].nic.baseband::L1_DsssBaseband:noOfRfWavesRxed = 69395
sim.host[10].nic.baseband::L1_DsssBaseband:noOfF1Errors = 0
sim.host[10].nic.baseband::L1_DsssBaseband:noOfPlcpErrors = 0
sim.host[10].nic.baseband::L1_DsssBaseband:noOfCrc32Errors = 0
```

Many RF waves do not generate Cas alarms but the frames coming in are free from bit-errors. The counter *noOfRfWavesRxed* is incremented even though a radio is in a transmitting state since the RF wave is present on the coax input port. Before leaving this section we present the network

capacity for the three capture models when the offered traffic employs all the priority levels. As seen by Figure 9.10, the normal capture model gives a capacity close to the perfect capture model. However, the zero-capture model, which does not tolerate any overlapping transmission, has a much lower capacity. The capacity of the latter may be improved by increasing b_p.

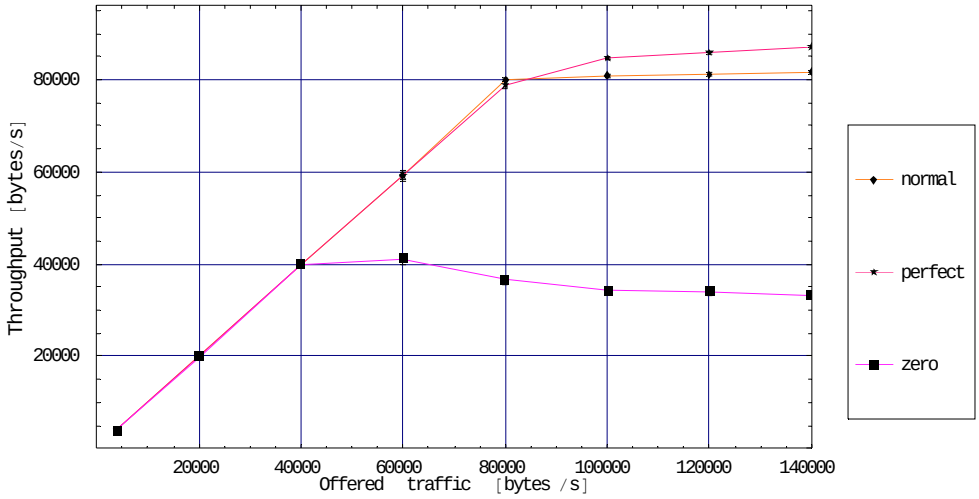


Figure 9.10 Simulated network throughput using multiple priority levels. The priority distribution is {P0,P1,P2,P3}={0.4,0.3,0.2,0.1}. Radio version 1Mbps.

9.5 Capacity versus network size

Up to now, we have optimised the b_p -set for a network containing 16 nodes. Consider a very large network. As the number of active nodes increases with increasing offered traffic, the network will reach a point where the collision rate leads to performance degradation. The MAC protocol would benefit from having an adaptive mechanism which increases the b_p -values under these conditions. Figure 9.9 illustrates different throughput courses versus offered traffic. Any system has a capacity limit and the curve marked as "perfect course" shows the idealized case. The worst situation is to have the instable course where the system collapses completely. We have experienced this situation when the backoff after CRC-error is removed. The network should be designed to have a graceful degradation as illustrated.

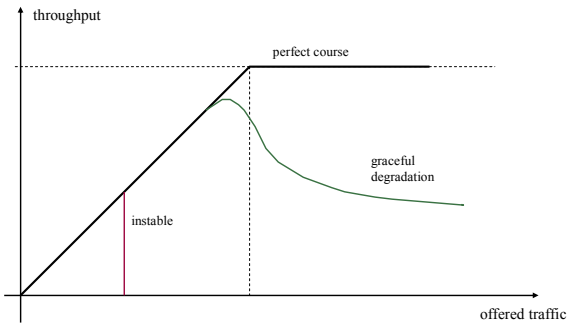


Figure 9.11 Illustration of possible throughput progress versus offered traffic.

Simulated throughput results for different network sizes are shown in Figure 9.12. The {9,16,25}-networks are all stable and the throughput-offered traffic relationship is nearly perfect. This in contrast to the {50,60}-networks that exhibit a graceful degradation due to a high collision rate. The cure against this is to increase the b_p-values but then smaller networks suffer from severe capacity degradation. A closer look at the {25}-network plot in the figure shows a small throughput drop from run 7 (120000 bytes/s) to run 8 (140000 bytes/s). Therefore we believe the b_p -values should be increased when the network size becomes larger than 30 nodes. The {3}-network operates at a very low collision rate but suffer from a long average channel idle period. Remember we have the backoff after transmission that also reduces the collision rate which obviously has a high impact in small networks.

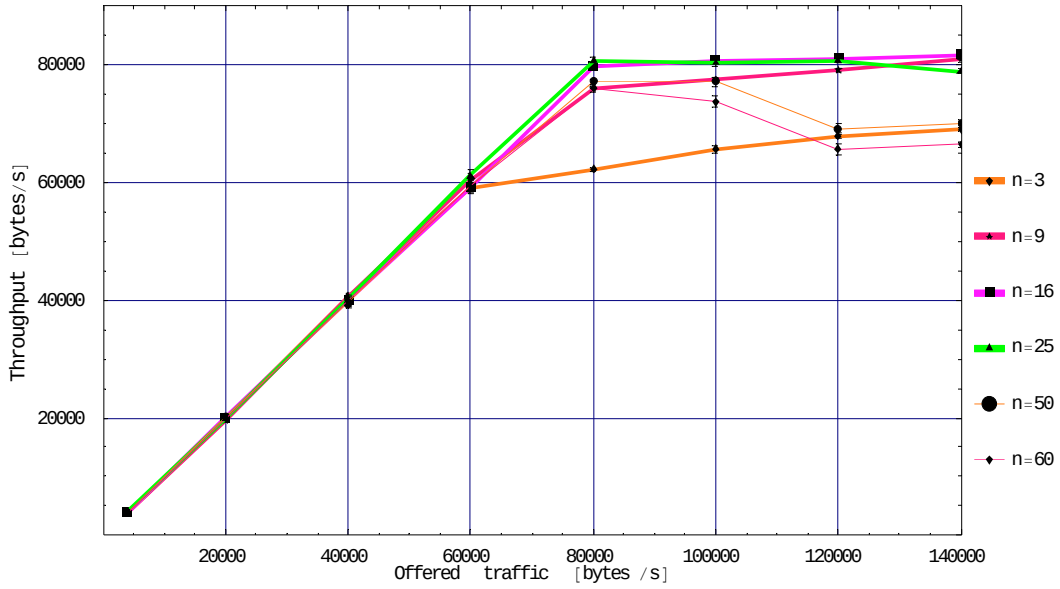


Figure 9.12 Estimated throughput using multiple priority levels. The priority distribution is {P0,P1,P2,P3}={0.4,0.3,0.2,0.1}. The legend expresses the network size as number of nodes. Radio version: 1Mbps. (simFeb2a).

Yet we have not validated the simulator for different network sizes. By using the single priority level P0 and setting the capture model to "perfect" for the scenario above, we have a network that confirms well to the theoretical throughput model described by equation (9.3) for large networks. Table 9.6 shows the deviation between simulated results and theoretical results. A perfect result is when the theoretical throughput falls within the simulated 90% confidence intervals. The deviation expresses how far outside the confidence interval the theoretical value is.

As expected we have a significant deviation for the 3-node network where the backoff after transmission leads to enlarged mean channel idle period not included in the theoretical model. For network size larger than 9-nodes, we achieve a high conformity between simulated and theoretical results. Note the high accuracy[29] of the simulated results which are better than 1%.

[29] half_width/mean

Network size	Simulated	Theoretical model	Deviation [%]
3 nodes	48538 ± 296	54183	11
9	70338 ± 302	71112	0.7
16	77710 ± 222	77713	0
25	81872 ± 163	81445	-0.3
50	86015 ± 145	85232	-0.7
75	87540 ± 142	86609	-0.9
100	88001.5 ± 151	87321	-0.6
150	87865 ± 169	88050	0
200	87325 ± 170	88420	1.1

Table 9.6 Throughput capacity [bytes/s] as function of the network size. Perfect capture and single priority level P0. Otherwise identical to the scenario in Figure 9.12. (simFeb2b)

9.6 Summary

By defining a simplified operating scenario and using the perfect capture model, we got a system that can be described analytically. We executed a number of experiments under these simplified conditions and compared the simulated results against analytical results. The validation phase discovered many errors that were corrected and new tests were done. Finally we got excellent conformity between the theoretical and simulated results. The scenarios developed in this chapter can serve as regression tests after modification and enhancements of the software.

10 Multihop Networks

This chapter deals with multihop networks and does a comparative analysis between different scenarios. When we turn to multihop cases, we no longer have any mathematical model that can give us an indication of the correctness of the simulation results. Therefore the strategy used is to gradually increase the complexity. The first section considers a multihop network where all the network nodes are within the same radio coverage area[30]. Here the MAC protocol prevents collisions effectively while the overhead caused by the 3a layer protocol is included. Section 10.2 takes this scenario as input and reconfigures the pathloss matrix to have disjoint radio coverage areas. Mobility is an important issue in ad-hoc networks. As mobile users move around, the network capacity becomes a function of time. Section 10.3 considers a complex scenario where the capacity of different user applications is measured as the nodes change their locations on the playground. The first sections use a fixed pathloss model, we get sharp radio ranges since the link cost values are either one or infinite (no connection). Section 10.4 studies a multihop network using the Egli pathloss model and the link cost values are quite different on the radio links. The maximum packet lifetime (see section 3.6) is set to 60 seconds in this chapter.

10.1 The Cost of Multihop Communications

The aim of this section is to illustrate how much transmission capacity multihop traffic consumes compared to single-hop traffic. We specify a beneficial scenario without hidden nodes but include the effect of the 3a level store-and-forwarding. This is accomplished by conducting the following two steps:

1) Specify a fixed pathloss matrix and set the transmitting power sufficiently high to have 20 dBm receiving power over the entire playground.
2) Specify a routing matrix which forces the system to use multiple hops despite the fact that the end destinations are within radio range.

Under these conditions the network operates as a multihop network, but the MAC protocol prevents the hidden-node problem and operates at low collision rate. Section 10.2 takes this example a step further by analysing the effect of hidden nodes. We use a single level priority P0, a fully connected network, the 100kbps radio version, the normal capture model and 400 bytes fixed payload size at layer 7. The traffic pattern and the routing matrix used are specified in Figure 10.1. Note the special traffic pattern and that the relay nodes N10 and N11 do not generate fresh traffic. If we use an “all-to-all” traffic pattern, we would see no or little difference in the network throughput because the throughput statistics would be overwhelmed by samples from single-hop traffic streams. The increase in the average channel idle period caused by the pacing protocol will be grabbed by the single hop traffic.

[30] The pathloss is set to 10dB for all links while the routing matrix forces the nodes not to use shortest path.

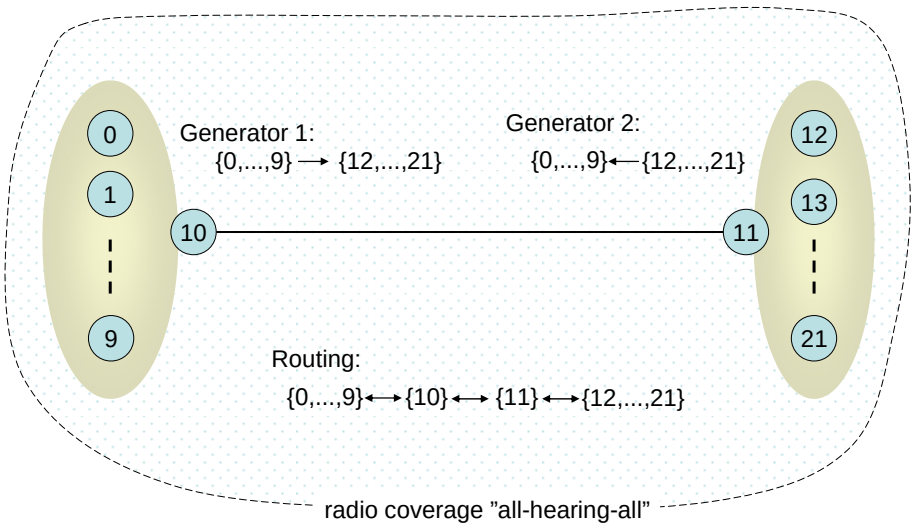


Figure 10.1 A fully-connected network containing 22 nodes where the traffic is forced to traverse 3-hops.

The forced idle time after transmission is not actually needed here. The system uses low priority traffic only and the MAC protocol then introduces a sufficiently long channel sensing period to keep the network stable. Therefore, the fixed backoff is set to zero in this section.

The single-hop system that serves as the reference network in this section also uses the end-to-end traffic pattern in Figure 10.1, but the routing matrix is modified to give a single-hop network. Hence N10 and N11 never send packets with this routing matrix. In summary, we have defined the following four scenarios to be included in this experiment:

Case **A1**: ARQ and one hop
Case **A3**: ARQ and three hops
Case **B1**: No use of ARQ and one hop
Case **B3**: No use of ARQ and three hops

Given the traffic matrix and the routing matrix above, N10 and N11 operate solely as relay nodes and layer 3a ARQ is always used on the link $N10 \leftrightarrow N11$ for case A3. ARQ at layer 2 will never be used on that link.

Figure 10.2 shows the simulated results. First we consider the capacity degradation to be expected going from a single-hop network to a 3-hop network. Since the traffic traverses three hops, we should immediately expect 2/3 capacity reduction compared to the one hop cases. Based on the B1 simulated results, the performance of the B3 case should be lower than $8000/3 = 2667$ bytes/s. Maximum throughput for case B3 is measured to 1718 bytes/s which represents 64% capacity reduction. The throughput drop after run number 5 ($\Lambda = 2000$ bytes/s) is mostly caused by packet lifetime expiry within the relay nodes. Queues build up in the relay nodes while the queues at the originating nodes remain short. The end-to-end delay is approximately 4 seconds in run number 5 and became 50 seconds in the next run ($\Lambda = 2400$).

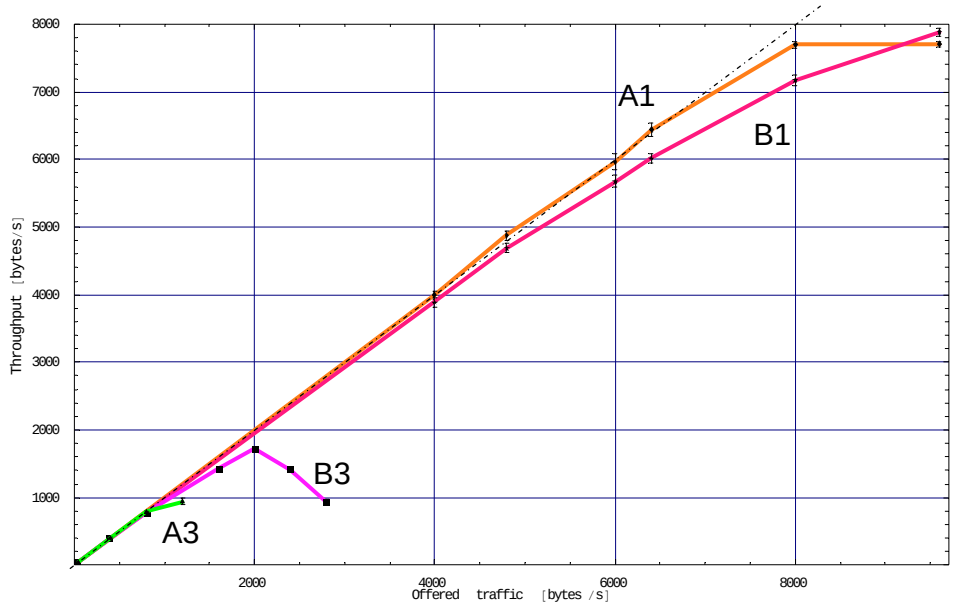


Figure 10.2 Simulated throughput versus offered traffic for the cases A1, A3, B1 and B3. 90% confidence intervals (simFeb17d).

B3 does not use ARQ and hence the pacing delay is not applied to the traffic. This in contrast to case A3 where the pacing protocol adds a 0.9 second delay already at $\Lambda = 800$. This chokes down the MAC offered load and collisions rarely happen in this network. Maximum throughput for A3 is measured to approximately 935 bytes/s, a 88% capacity reduction compared to case A1. The measured number of retransmissions per packet delivered was estimated to 0.23 ± 0.02 for case A3, which is higher than expected under low collision rate. The retransmission timer at layer 3a is set to 2 *pacing and this seems to be too short for A3. The timer times out too early and Figure 10.3 shows that it is beneficial to increase the timeout interval.

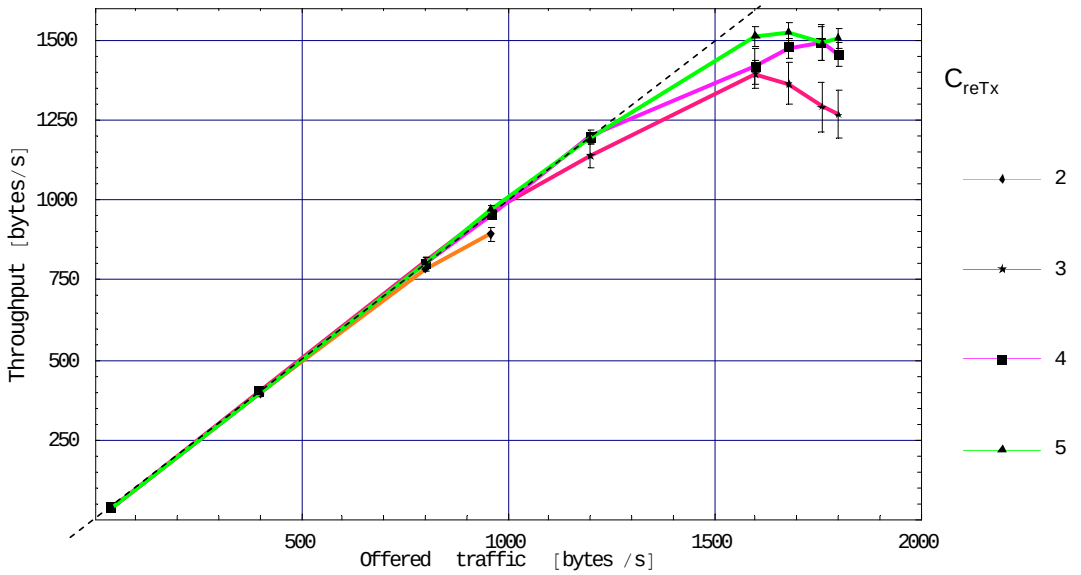


*Figure 10.3 Case A3 throughput capacity for different multiplying constants C_{reTx}. The 3a layer retransmission timer is set to C_{reTx} * pacing.*

10.2 Network Fragmentation

As the connectivity of the fully connected network studied in the previous section deteriorates, the MAC protocol is unable to regulate access to the channel in an orderly manner due to hidden nodes. The objective of this section is to study the performance when the network suffers from the hidden-node problem. The only deviation between this section and the previous section is the pathloss matrix, which is modified to split the two 10-node groups into two none interfering radio coverage areas, see Figure 10.4. Node N10 and N11 operate as relays and experience identical traffic conditions.

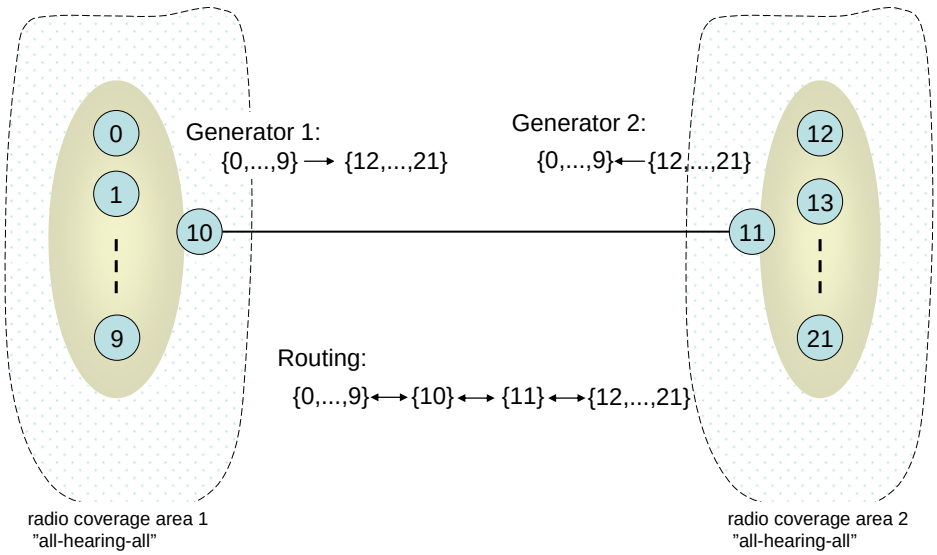


Figure 10.4 The scenario is divided into three radio coverage areas. Two regions contain 11 nodes and one region contains the two nodes {10,11}.

Let $H_{i\to j}$ denote the hidden-node set for the link from node *i* to node *j*. $H_{0\to 10}=\{11\}$ and $H_{10\to 11}=\{12,...,21\}$ are two example sets for the current scenario. The link $0\to 10$ is sensitive to the load level at N11 while the passive acknowledgement $10\to 0$ seldom fails to reach N0 since the MAC protocol works efficiently among the nodes $\{0,...,9,10\}$. The forwarding of packets on the link $10\to 11$ may often fail since N11 has many hidden-nodes. Likewise, the passive acknowledgement $11\to 10$ may experience a high likelihood of being hit by a transmission from one of the nodes in the set $\{1,...,9\}$ (assumes N0 is the originator and its pacing timer is running).

A successful packet transmission from N10 to N11 depends on two events: 1) N11 must be in the preamble search state and 2) The number of overlapping transmissions must be less or equal one. One hit gives 0 dB SNR that again gives the demodulation success rate $0.999^{422}=0.66$ while two hits give -3 dB SNR and the success rate $0.85^{422}=0$. See appendix A.

Figure 10.5 shows the simulation results where the 3-hops plots from Figure 10.2 have been included.

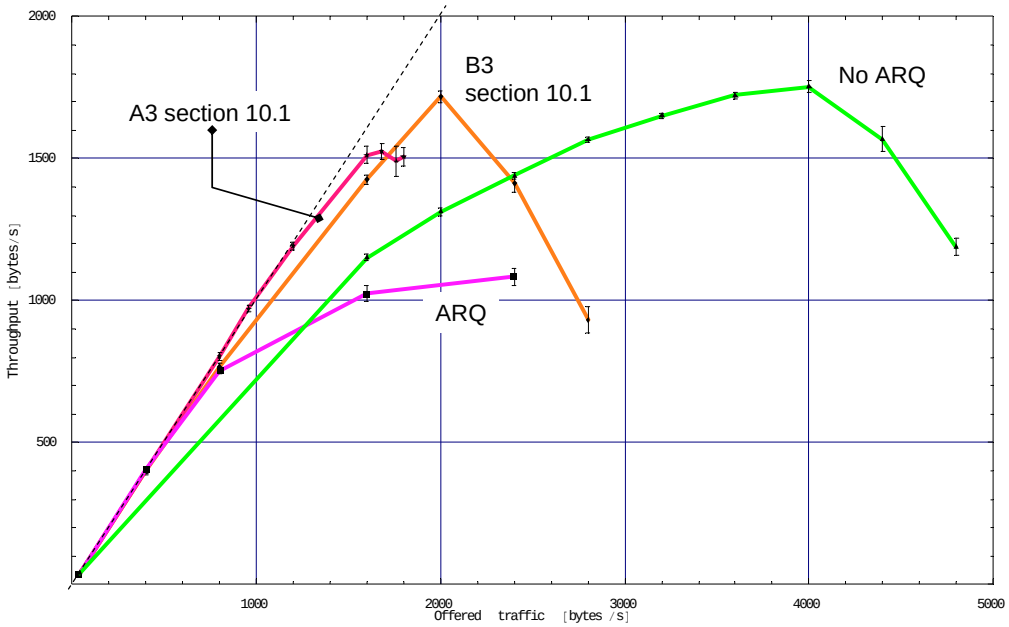


Figure 10.5 Simulated throughput versus offered traffic using a two-way traffic pattern between the two 10-node groups (simFeb23a).

Firstly, we look at the system when ARQ is not activated and note that the maximum throughput (1752 bytes/s) is of the same magnitude as in the previous section. More MAC transmission capacity becomes available in this section because the two 10-nodes groups have separate radio channels and can feed their relays faster. However, the end-to-end delay has increased from 4 seconds to 12 seconds[31]. Also note the increase in packet loss probability which has increased from approximately 14% $(1-1718/2000)$ to 56% $(1-1752/4000)$. The hidden-node effect increases the packet corruption rate in the order of 10% and only N10 and N11 show a significant corruption rate. The major cause of loss is lifetime expiry.

The capacity for ARQ traffic drops from 1500 bytes/s in section 10.1 to 1100 bytes/s when hidden-nodes are introduced. This represents 26% performance loss.

10.3 Mobility

The purpose of this section is to illustrate how *oTWLAN* can be used to estimate performance for mobile users. We consider a hypothetical scenario where 16 users move while one stays at a fixed location to maintain connectivity to a head quarter located somewhere in a backbone network.

A Battlefield Management System (BMS) provides location of friendly forces in the combat arena. The BMS tracks the soldiers within the group but also gives information from higher operating levels, e.g. a common operating picture. The former service is named "BMS-internal" and the latter "BMS-global". BMS-global is not considered to be mission critical and is therefore assigned the lowest priority level (P0). The group coordinates their activities through an internal message system and this application is assigned priority level P2 since it is mission critical. The

[31] Confidence control has not been applied to delay measurements and they might be inaccurate.

group may receive important alarm and intelligence (AI) information from the head quarter. This application sends packets at a low rate. It is assigned the highest priority level because the information content is important and must be given precedence at the network level over other traffic streams. Table 2.1 specifies the relative traffic volume for each application type and the layer 7 payload size distribution. The 100kbps radio version is used in this section and the applications enable use of the ARQ protocols at the network level.

Application type	Priority	Percentage share	Payload size [bytes]
Alarm and intelligence	P3 (highest)	5%	fixed 100
Internal message exchange	P2	35%	fixed 100
BMS internal	P1	10%	rand(50,400)
BMS global	P0 (lowest)	50%	rand(50,400)

Table 10.1 Applications types and traffic parameters. Payload size refers to layer 7. "rand(50,400)" means randomly distributed in the integer range r, $50 \le r \le 400$. *The "percentage share" column gives the traffic volume per application relative to the network offered load in packets per second.*

The initial position is shown in Figure 10.6. All nodes are within the radio coverage area of each other at time instance t_0. As they move towards the target area at the lower right corner of the playground, they experience changing radio coverage caused by a varying terrain profile. The group splits into two equally sized groups when they reach the hill. At time instance t_1 (Figure 10.7), node N1 and N9 stays behind to keep the two groups connected but also to maintain the radio connection to N0. The traffic pattern is specified in Table 10.2. Note the AI traffic and the BMS-global traffic which are one-way, and that the originators are always attached to the Wide Area Network (WAN).

Application type	Time instance t_0	Time instance t_1 and t_2
Alarm and intelligence (AI)	$0 \rightarrow \{1,...,16\}$	$0 \rightarrow \{2,...,8\} \cup \{10,...,16\}$
Internal message exchange (IME)	$\{1,...,16\}$	$\{2,...,8\} \leftrightarrow \{10,...,16\}$
BMS-internal	$\{1,...,16\}$	$\{2,...,8\} \leftrightarrow \{10,...,16\}$
BMS-global	$0 \rightarrow \{1,...,16\}$	$0 \rightarrow \{2,...,8\} \cup \{10,...,16\}$

Table 10.2 Application types and traffic pattern.

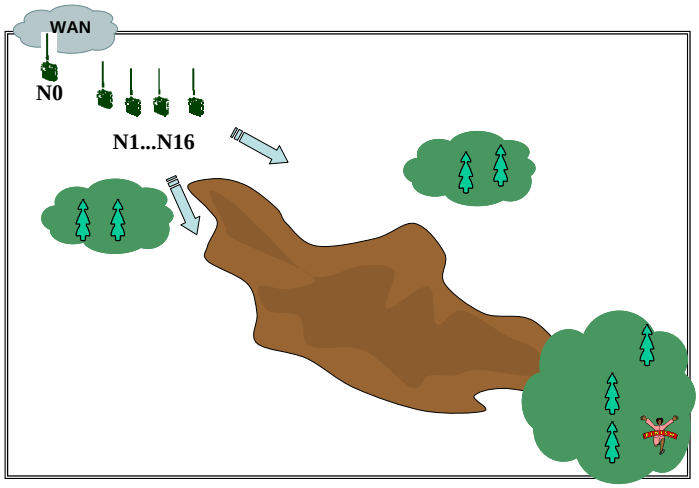


Figure 10.6 Playground layout at time instance t_0. All the network nodes are within the same radio coverage area. N0 operates as a gateway to the WAN.

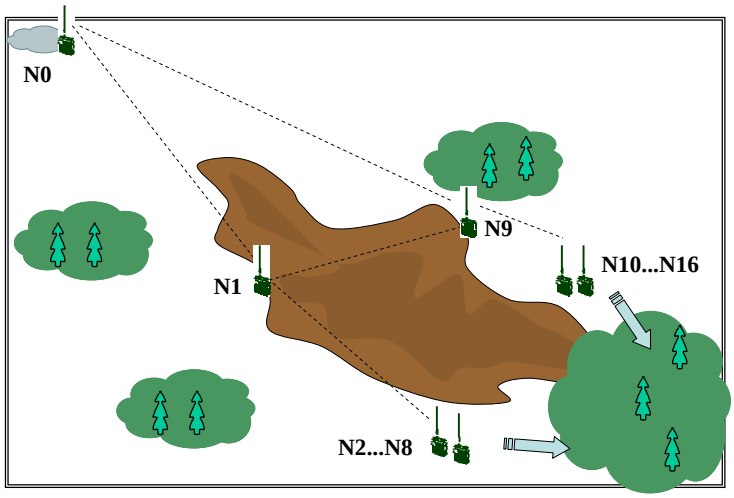


Figure 10.7 Playground layout at time instance t_1.

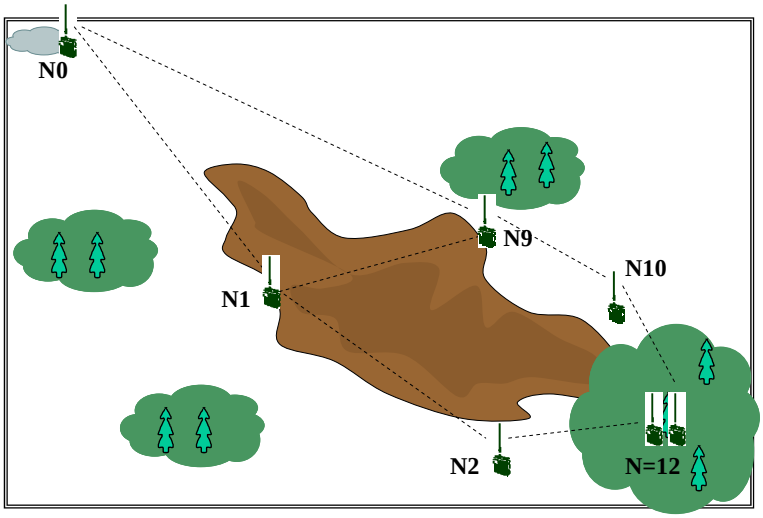


Figure 10.8 Playground layout at time instance t_2.

Figure 10.9 presents the simulated results for time instance t_0. The relative traffic volume per application refers to packets per second, but since the payload size distributions are different, the relative traffic volume in bytes per second is different. When the network load level is 1 packets/s, the offered traffic in bytes/s is 175, and we have the traffic volume in bytes/s relative to the offered traffic Λ [bytes/s]:

$$\{0.05, 0.35, 0.1, 0.5\} \bullet \{100, 100, 225, 225\} / 175 = \{0.029, 0.200, 0.129, 0.643\}.$$

The set ordering is P3...P0.

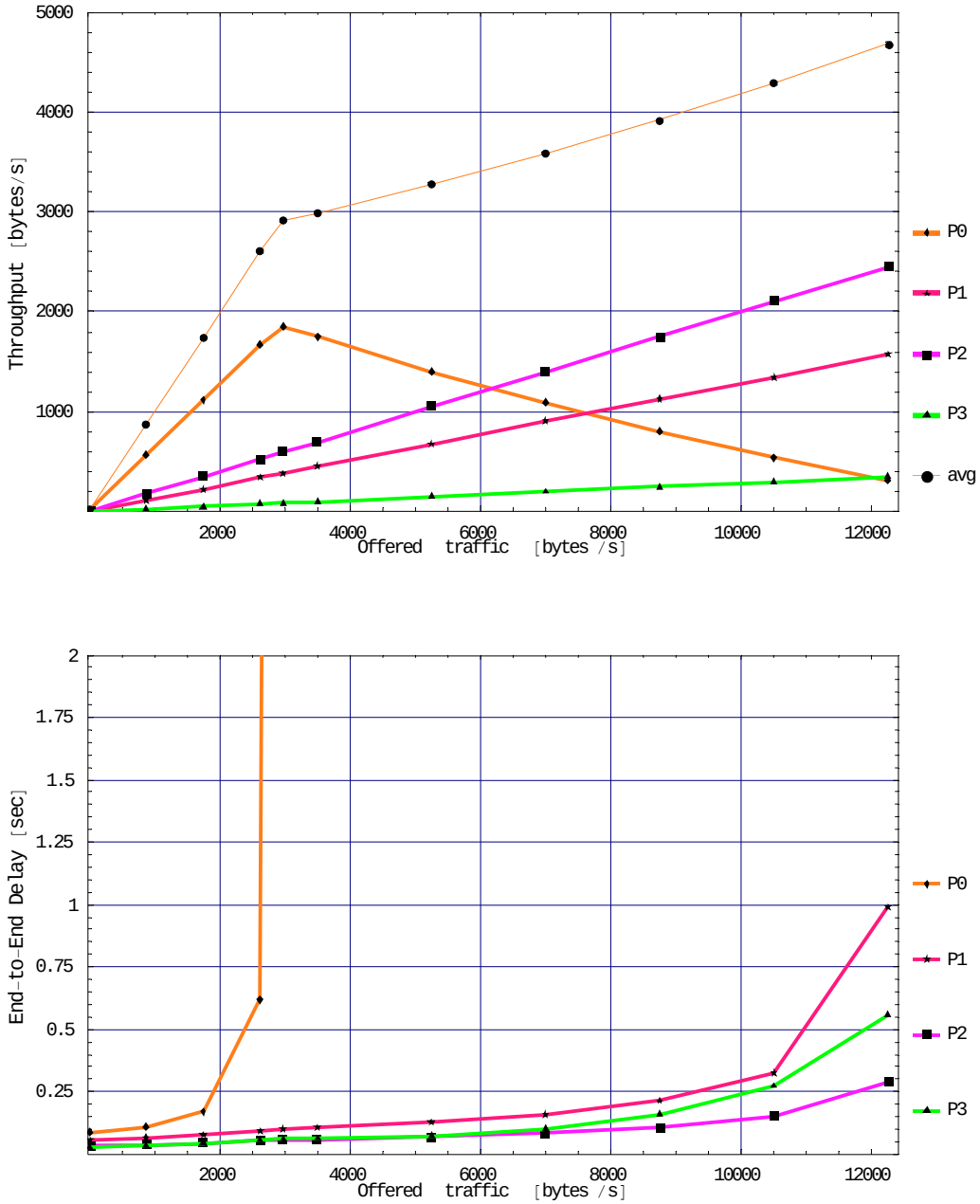


Figure 10.9 Simulated performance for time instance t_0. Confidence control is only applied to the cumulative throughput plot.

Network saturation performance is defined as the measurements taken at a traffic level where the network **starts** to drop packets due to an overload condition. The network invokes the precedence function (MLPP) when the offered traffic increases beyond this level. The lower level priority traffic experience longer end-to-end delays and the packet loss rate due to lifetime expiry increases.

The network *capacity limit* is defined as the load level at which the network starts to drop mission critical traffic which means priority level P1 in this section. TCP-based applications retransmit lost packets and the increased end-to-end retransmission rate normally leads to exponentially increased offered traffic.

The maximum throughput for the BMS-global application (priority P0) is 1850 bytes/s, see Figure 10.9. However, the delay plots show that the delay increases rapidly at $\Lambda = 2625$ and the throughput capacity for this application $\lambda = 0.643 \cdot 2625 = 1687$ bytes/s in practise. Note the elapse of the P3 delay curve that exhibits a higher delay than lower priority traffic P2. Here we see the effect of the forced backoff after transmission which has great impact since the network has one single node sending this priority level. The network saturation performance is:

AI traffic (P3):	76 bytes/s	@ 0.05 sec[32]
IME (P2):	525 bytes/s	@ 0.05 sec
BMS-internal (P1):	338 bytes/s	@ 0.1 sec
BMS-global (P0):	1688 bytes/s	@ 0.6 sec
Cumulative P0...P3:	2625 bytes/s	@ 0.3

The throughput plot shows a linear progress for the priority levels P1...P3 and we cannot deduce a capacity limit from the plot. However, the delay plot shows a fast increasing delay for P1 traffic at the highest load level, and thus we assume that the capacity limit for this priority level is approximately $\lambda = 0.129 \cdot 12250 = 1580$ bytes/s. The performance measured at this point is:

AI traffic (P3):	355 bytes/s	@ 0.6 sec
IME (P2):	2450 bytes/s	@ 0.3 sec
BMS-internal (P1):	1580 bytes/s	@ 1.0 sec

The network serves multihop traffic at time instance t_1 and the routing matrix used specifies the following routing paths:

AI and BMS-global traffic:

$$N0 \rightarrow N1 \rightarrow \{N2,...,N8\} \text{ and } N0 \rightarrow N9 \rightarrow \{N10,...,N16\} \quad \text{2-hops}$$

IME and BMS-internal traffic:

$$\{N2,...,N8\} \leftrightarrow N1 \leftrightarrow N9 \leftrightarrow \{N10,...,N16\} \quad \text{3-hops}$$

[32] Notation is throughhut@end-to-end-delay

The relay nodes N1 and N9 are no longer included as end-destinations as they were at time instance t_0, see Table 10.2. N1 and N9 are relay nodes so the traffic is passing by anyway and we do not mix single hop statistics with multihop statistics. This should give a more realistic picture of the multihop traffic capacity.[33]

The BMS-global traffic reaches its maximum at $\Lambda = 1000$ and drops rapidly as the offered traffic increases. The delay plot shows that the end-to-end delay becomes high above this level and we estimate the maximum throughput capacity for the BMS-global application to be approximately $\lambda = 0.643 \cdot 1000 = 643$ [bytes/s]. The throughput curve for the BMS-internal traffic (P1) has a flat course and peaks at $\Lambda = 1925$ giving the P1 throughput $\lambda = 0.129 \cdot 1925 = 171$ [bytes/s]. However, many packets are lost due to lifetime expiry at this load level. The loss starts already at $\Lambda = 962$ for P0 traffic and $\Lambda = 1225$ for P1 traffic.
We conclude that the network saturation performance is:

AI traffic (P3):	28 bytes/s @ 0.4 sec
IME (P2):	192 bytes/s @ 1.7 sec
BMS-internal (P1):	124 bytes/s @ 2.0 sec
BMS-global (P0):	618 bytes/s @ 2.2 sec
Cumulative P0...P3:	962 bytes/s @ 1.9

The capacity limit is estimated to:

AI traffic (P3):	35 bytes/s @ 0.7 sec
IME (P2):	245 bytes/s @ 2.8 sec
BMS-internal (P1):	158 bytes/s @ 3.3 sec

The two groups move into the forest at time instance t_2 with the exception of N2 and N10. They stop on the outskirts of the forest and are given the task to maintain the backward radio connectivity. Moving into a forest always means a dramatic reduction of the radio coverage area. The intention of this section is not to end up with a discussion of radio range and transmission capacity in a forest. Therefore we assume the group operates close together such that the radio connectivity among the nodes $\{3,...,8\} \cup \{11,...,16\}$ remains good. The radio links $N2 \rightarrow \{3,...,8\}$ and $N10 \rightarrow \{11,...,16\}$ are also assigned optimistic pathloss values (10dB).

Figure 10.11 presents the simulated results for time instance t_2. The BMS-global traffic (P0) reaches its maximum at the offered load $\Lambda = 1050$ bytes/s given the maximum throughput 445 bytes/s for this application. However, the packet loss starts already at $\Lambda = 262$ and we conclude that the network saturation performance is:

[33] The simulator has not yet implemented probes for collecting statistics per node. Applying confidence control of such probes may lead to impractical long run-times.

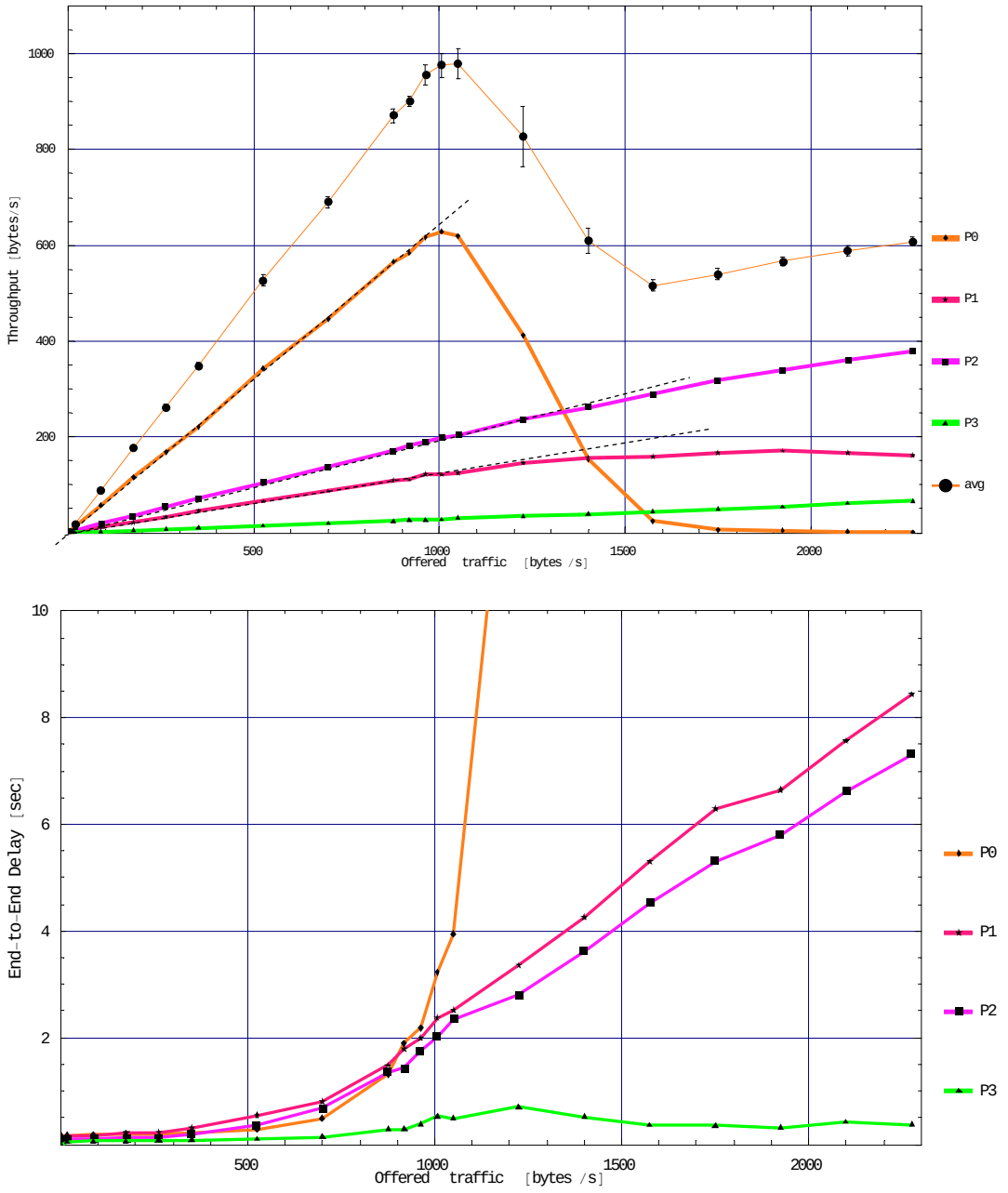


Figure 10.10 Simulated performance for time instance t_1. Confidence control is only applied to the cumulative throughput plot.

AI traffic (P3):	7 bytes/s	@ 1.2 sec
IME (P2):	53 bytes/s	@ 0.04 sec
BMS-internal (P1):	34 bytes/s	@ 0.06 sec
BMS-global (P0):	168 bytes/s	@ 3.8 sec
Cumulative P0...P3:	262 bytes/s	@ 2.0

Both the IME traffic (P1) and the BMS-internal traffic (P2) have low delays because most packets traverse only one hop. Only transmissions between N2 and N10 take two hops but are of course invisible in a network average estimate. The low foliage loss and the traffic pattern give an

insignificant loss of the P1 and P2 traffic and the capacity limit cannot be deduced from the results. P3 also has insignificant loss but a much higher delay because the packets traverse two or three hops. The special form of the P3 delay curve is caused by the adaptive pacing delay.

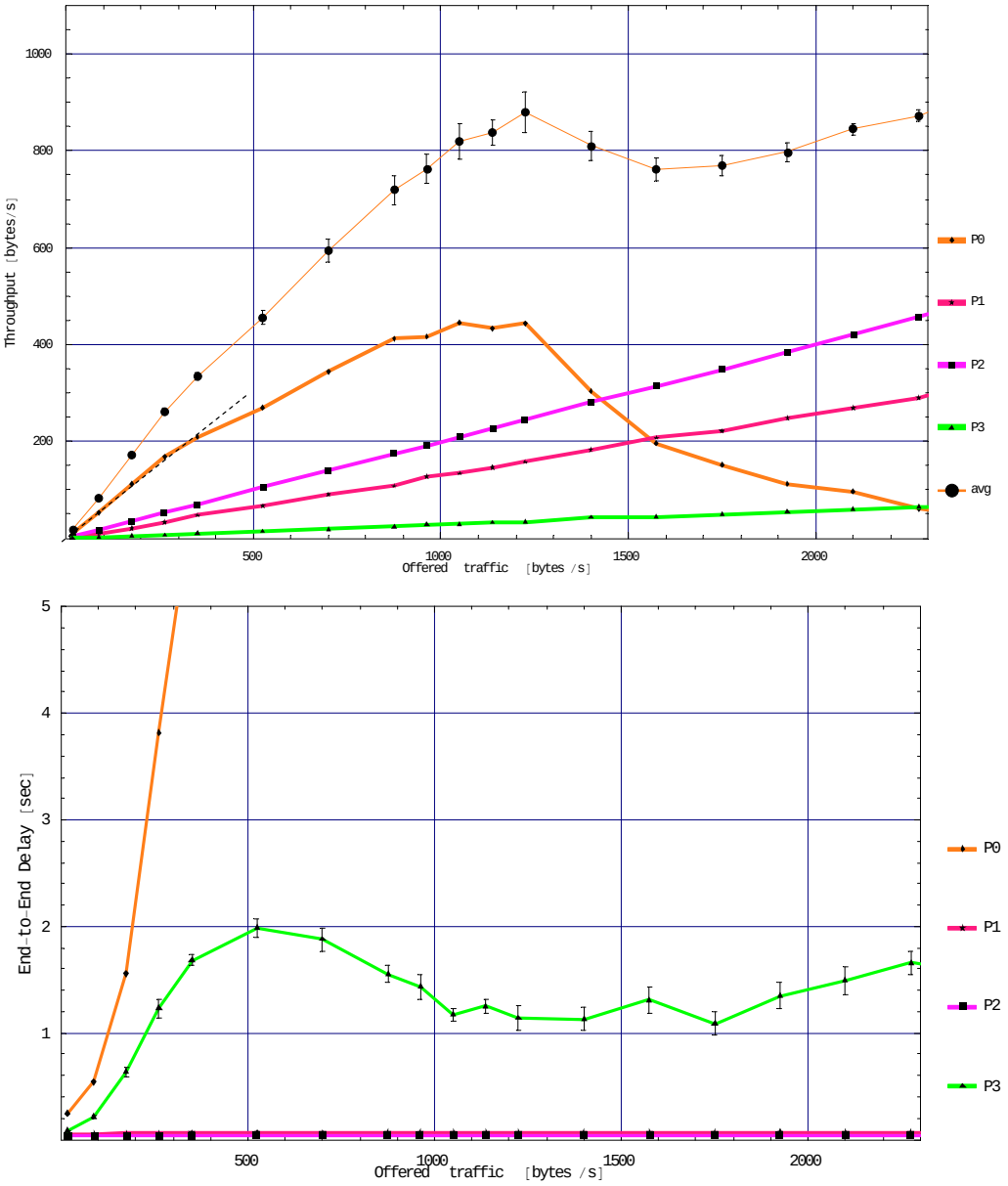


Figure 10.11 Simulated performance for time instance t_2. Confidence control is only applied to the cumulative throughput plot.

10.4 Multihop in Egli terrain

This section uses the Egli pathloss model and studies a network containing 25 nodes located on the regular grid shown in Figure 10.12. The network connectivity is changed by altering the transmitting power and the only parameter that we modify between the nodes is the node position on the playground. The preamble and the payload field in the radio frame have different SNR characteristics and hence different ranges. These ranges as function of the radiated power and the radio parameters used are stated in Table 10.3.

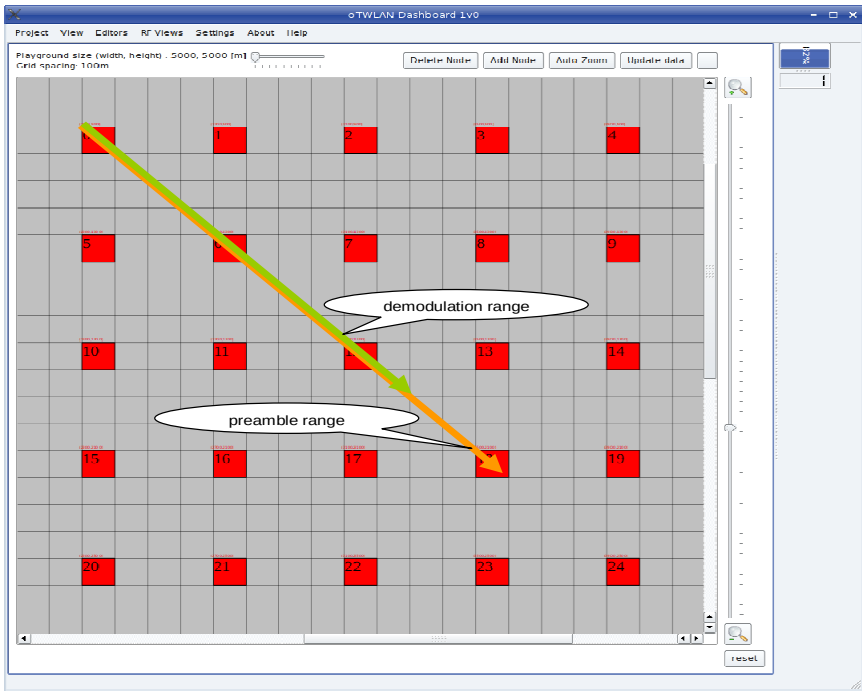


Figure 10.12 The playground layout with grid spacing 100 meters. The node positions are the upper left corners of the red boxes. The arrows indicate the preamble range and the demodulation range for node 0 when it sends at -10 dBm.

A

Transmitting power [dBm]	Preamble range [m]	Demodulation range [m]
10	5846	4512
0	3287	2537
-2	2929	2260
-6.5	2251	1737
-10	1848	1426

B

Parameter	Value
Radio hardware	100kbps version
Antenna height	2m
Antenna gain (tx/rx)	0 dBi
Pathloss model	Egli

Table 10.3 Radio coverage as function of the transmitting power (table A) using the radio parameters in table B.

The longest link in the network is 2262 meters and when transmitting at 10dBm all nodes are reached under high SNR levels during low traffic periods. Table 10.4 expresses the link cost

values versus the transmitting power seen from node *N0*'s point of view. The infinite sign signifies that the destination is outside the radio range. The number of usable radio links is expressed in the column "degree". Remember that the link cost range is $1 \le r \le 1.49$ where 1 is a high quality link (SNR > +3dB). Section 5.4 specified the lower SNR threshold for a link to be -6dB. The link cost is determined at time instance zero and is based on the background noise only. As the traffic increases, the SNR decreases due to collisions.

Tx power [dBm]	**N1**	**N3**	**N4**	**N9**	**N12**	**N13**	**N14**	**N17**	**N18**	**N19**	**N22**	**N24**	**degree**
10	1	1	1	1	1	1	1	1	1	1	1	1	24
0	1	1	1.05	1.08	1	1	1.16	1	1.11	1.26	1.16	1.38	24
-2	1	1	1.16	1.19	1	1.06	1.27	1.06	1.22	1.37	1.27	∞	23
-6.5	1	1.14	1.41	1.44	1.08	1.31	∞	1.31	1.47	∞	∞	∞	19
-10	1	1.33	∞	∞	1.27	∞	∞	∞	∞	∞	∞	∞	12

Table 10.4 Link cost values from node N0 to other nodes at different transmitting power levels.

A new routing table is calculated for each power level. However, we have a single-hop network at 10dBm and 0dBm. All nodes are reached within two hops under the other power levels.

This is a heterogeneous network and is therefore difficult to analyse. The edge nodes {0,4,20,24} operates under more severe RF conditions than the centre node *N12*. The MAC protocol relies on correctly detecting the channel state busy/idle. However, nodes failing to detect the preamble correctly, assume the channel is idle and may transmit. The result is lower SNR in the network. Nodes detecting the preamble but operating near the demodulation range, behave differently. They frequently experience CRC errors and enter the forced idle state. They are prohibited to transmit for a period of time or until a successful packet receipt has occurred.

The layer 7 traffic data for this scenario is presented in Table 10.5. *randInt* is a uniform random distribution in the integer range $50 \le r \le 400$, that is, the average layer 7 payload size is 225 bytes.

Parameter	**Value**
Message arrival distribution	exponential, variable mean
Payload distribution	randInt[50, 400] bytes
Pattern	uniformly distributed
Priority distribution	Single level P0(lowest)

Table 10.5 Traffic data for user terminals (offered traffic to layer 7).

Figure 10.13 and Figure 10.14 present the simulated throughput results for the different power levels. The throughput curve shows significant performance degradation when the power level drops to -2dBm even though we have a good connectivity; the nodal degree for the corner nodes are 23. The majority of the packets reach their destinations in one hop. The challenge in this

network is the high interference level that occurs with increased traffic. The MAC protocol fails to coordinate the access to channel as more hidden nodes are introduced. A further discussion of this example is done in appendix B.

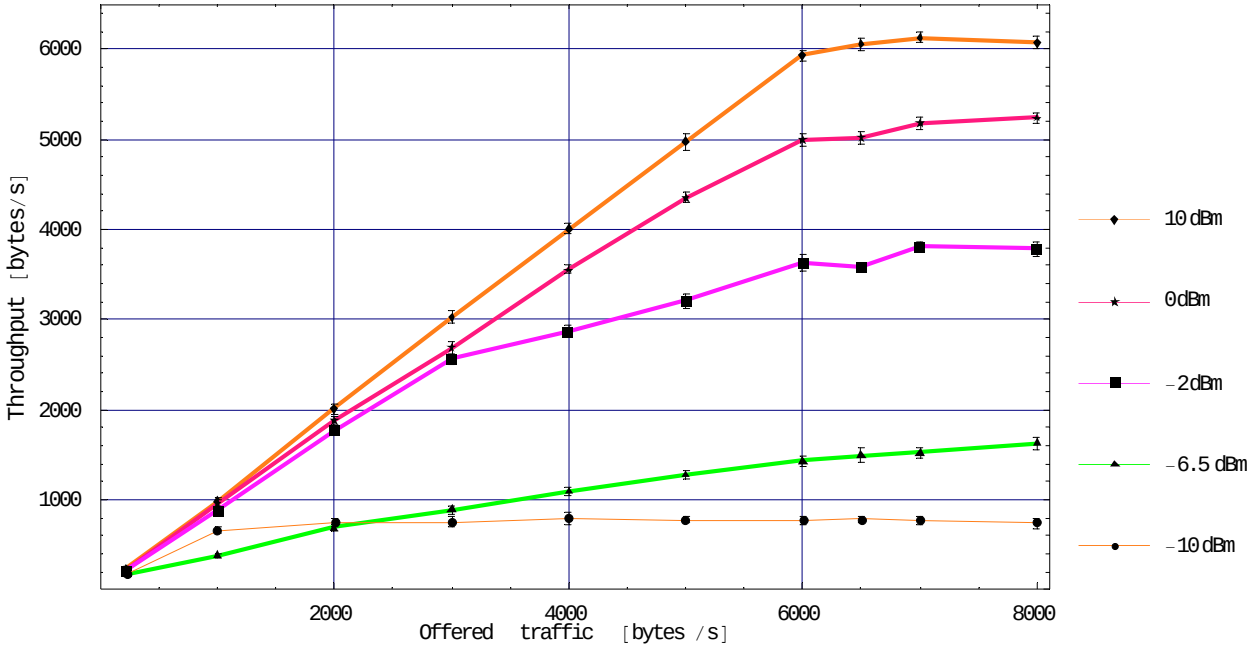


Figure 10.13 Throughput versus offered load.

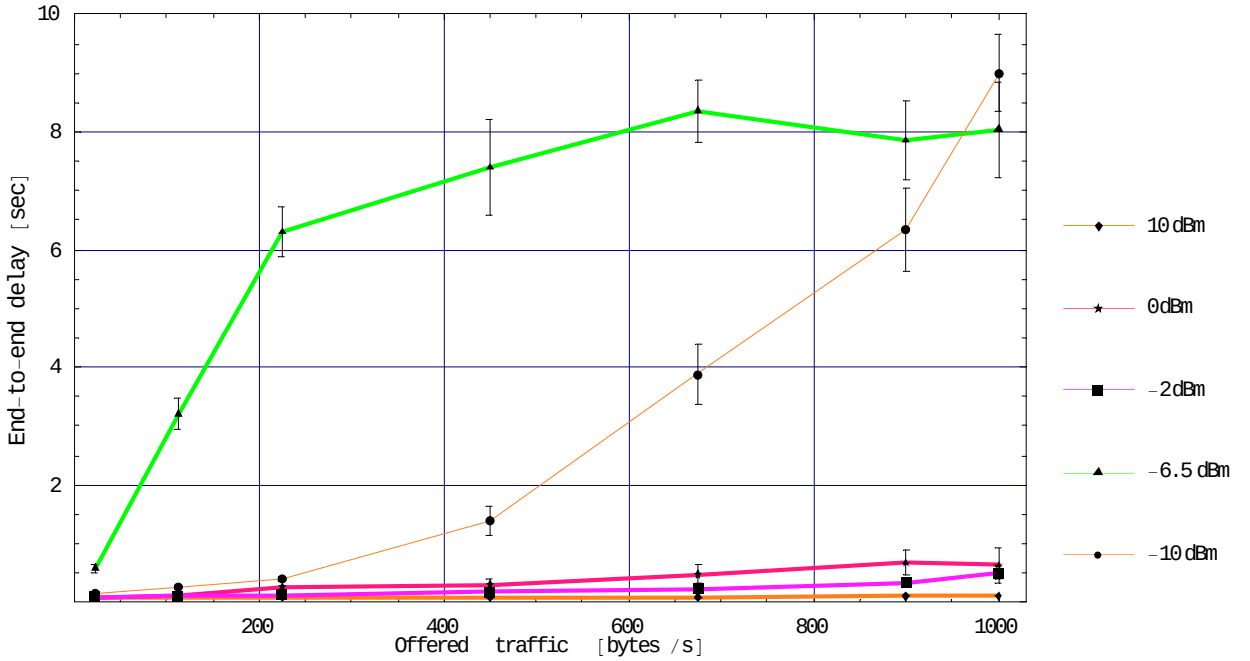


Figure 10.14 End-to-End delay versus offered traffic.

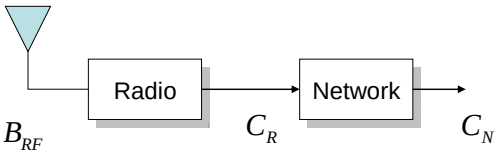
10.5 Discussions and Conclusions

This chapter has estimated the performance of multihop communication and showed that the throughput capacity drops dramatically compared to single-hop communication. It is the 3a layer pacing protocol that serves multihop traffic and thus the protocol has a great impact on the performance. We do not argue that this protocol is the most efficient protocol compared to other protocols (e.g. the RTS/CTS protocol). However, we believe it has reasonable efficiency compared to other solutions.

TCP traffic streams must be served by an ARQ protocol on each radio hop to avoid end-to-end retransmission at the TCP level. Therefore the ARQ simulation results are most interesting. Network efficiency[34] (Figure 10.15) is the ratio between the network level throughput [bytes/s] and the radio payload transmission rate [bytes/s][35]. The network efficiency for the experiments conducted in this chapter is:

1-hop traffic (section 10.1):	7650/12500 = 0.61
3-hop traffic without hidden nodes (section 10.1):	1500/12500 = 0.12
3-hop traffic with hidden nodes (section 10.2):	1100/12500 = 0.09

These numbers give an impression of the cost of serving multihop traffic. Even a protocol which handles the hidden-node problem perfectly gives significantly lower efficiency compared to single hop cases. An interesting research question is: shall we design a long range low capacity radio or a short range high capacity radio?



Radio efficiency: $\beta_R = \frac{C_R}{B_{RF}} \quad \frac{[bits/s]}{[Hz]}$

Network efficiency: $\beta_N = \frac{C_N}{C_R/8} \quad \frac{[bytes/s]}{[bytes/s]}$

Figure 10.15 Efficiency defined as the ratio between the payload capacity provided and the capacity "consumed". B_{RF} is the RF bandwidth occupied by the radio. C_R is the physical layer payload transmission rate. This should actually have been the physical layer throughput to include the overhead at the radio level such as the preamble. C_N is the network throughput.

[34] Often referred to as normalised throughput
[35] 12500 bytes/s for the 100kbps radio used in this chapter

Section 10.3 conducted a more practical study of multihop networks than the other sections. Table 10.6 summarises the results and presents an efficiency factor referred to the cumulative throughput for the single-hop case. We give a final observation about the pacing protocol. The pacing protocol showed to be an efficient flow control mechanism because the queue sizes increased at the 3a layer within the entry-nodes while the 3a layer transit nodes remain short as the offered load increases. Some similar experiments were conducted with pacing disabled and now the system suffered under long transit queues.

Application type	$\mathbf{t_0}$	$\mathbf{t_1}$	$\mathbf{t_2}$
Alarm and intelligence (P3)	76 bytes/s @ 0.05	28 bytes/s @ 0.5 sec	7 bytes/s @ 1.2
IME (P2)	525 bytes/s @ 0.05	192 bytes/s @ 2.0 sec	53 bytes/s @ 0.04
BMS internal (P1)	338 bytes/s @ 0.1	124 bytes/s @ 2.4 sec	34 bytes/s @ 0.06
BMS global (P0, lowest)	1688 bytes/s @ 0.6	618 bytes/s @ 3.2 sec	168 bytes/s @ 3.8
Cumulative P0...P3	2627 bytes/s @ 0.3	962 bytes/s @ 2.6	262 bytes/s @ 2.0
Efficiency	1	0.37	0.10

Table 10.6 Efficiency calculated from the network saturation performance.

11 Installation

This chapter explains how to install the *oTWLAN* project. A prerequisite to succeed is that the following software components have been installed on your computer:

- Qt version 4[36]
- the omnetpp version 3 including INET

The build process assumes the following default paths:

$OMNET	/usr/local/omnetpp-3.3
$OMNETI	/usr/local/INET-20061020
$OTWLAN	The directory where the *oTWLAN* tar ball is unpacked
$HOME	The login directory on Linux

The default path can easily be altered by editing the configuration files. The project is a *qmake* based project where the **.pro* files specify the software environment. If the omnet files are located in other directories, **make changes to the **.pro* files** to reflect your software environment. The file *$OTWLAN /otwlan/INSTALL* gives additional details. The installation steps are basically identical to any standard *qmake* based project:

cd *to_the_subproject_dir*; **qmake** *the_pro_file.pro*; **make** *clean*; **make**[37]

[36] The development was done under 4.3.1.
[37] A make distclean will also delete any existing Makefile

oTWLAN has three subprojects located under *inet*, *otcp* and *oprobe*. The directory *oprobe* contains the enhanced/modified *oProbe* source code. Be aware of that the *oTWLAN* main project (*$OTWLAN /otwlan/src*) is based on a number of subprojects and the build sequence is: *libinet.so*, *liboprobe2a.so, libotcp.so*. The classes needed from the *INET* project are linked into the library *libinet.so.*

The installation steps described herein have been tested on an "out-of-the-box" installation of *Mandriva 2008.0.* No testing has been conducted on other Linux variants, nor on Windows.

11.1 Basic Build

The basic build describes how to install *oTWLAN* without using the *oTCP* project. Follow the steps below.

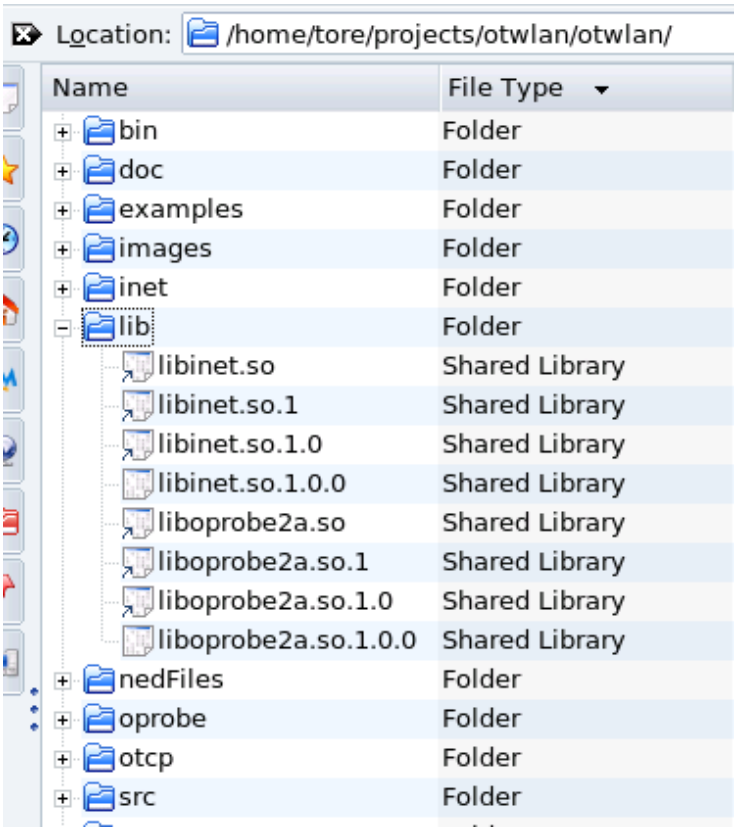
Step 1 *Unpack the tar file*

Go to the directory where you want to install the project ($OTWLAN) and then execute the command "tar xvfz otwlanV1.tar.gz". The simulator configuration files and source code files are extracted from the tar ball. Check that your directory structure looks like this:

+- doc/html	:	The location of doxygen documentation for the project. Top level is *index.html*.
+- inet	:	A subproject which compiles and links the INET objects required into the dynamic library *libinet.so*.
+- otcp	:	The *oTcp* project organised as a subproject which builds the DLL *libotcp.so*.
+- oprobe	:	The *oProbe* project organised as a subproject which builds the DLL *liboprobe2a.so*.
+- src	:	The source code area for the *oTWLAN* specific classes.

Step 2 *Build the libraries (libinet.so*, *liboprobe2a.so)*

The previous step placed the *oTWLAN* source code under *$OTWLAN/otwlan*. Use the basic *qmake* build sequence described above with one exception. When building you must issue a "qmake inetNoTcp.pro" to prohibit inclusion of the TCP related classes. After a successful build, check that the directory structure looks like this:



If not, this step failed and you have to search for error messages in the text stream. To bring the installation back to the initial stage, just type "make distclean" and the *Makefile* created by the *qmake* is also removed.

We have experienced that some Linux versions complain against unresolved symbols even though we build dynamic libraries. To solve this problem enable, the *QMAKE_LFLAGS* added at the end of all **.pro* files, by removing the starting "#"-sign.

Step 3 *Build bin/otwlan*

Do a "cd $OTWLAN/otwlan/src" and execute the sequence "qmake **otwlan.pro**; make clean; make".

Step 4 *Build Makefile at the top level*

Change working directory to $OTWLAN and do a *qmake otwlan.pro* at the project top level and then a *make* just to see that all the subprojects have been built successfully. **Warning**: If you execute a "make clean" at this level, all the subprojects are also cleaned!

Step 5 *Run-time link bindings*

Add the following line to the Linux system file */etc/ld.so.conf*:

$OTWLAN/otwlan/lib

Then make the change active by the Linux command *ldconfig.*

Step 6 *Start oTWLAN*
Please read section 11.3 before going on with this step. *oTWLAN* is started by executing *$OTWLAN/otwlan/bin/otwlan*.

11.2 Build with TCP Support

This section explains how to build *oTWLAN* with TCP support.

Step 1 *Unpack the tar file*
Same as step 1 in section 11.1.

Step 2 *Build the libraries (libinet.so, liboprobe2a.so, libotcp.so)*
Same as step 2 in section 11.1 with the exception that you must issue a ”qmake inet.pro” instead of ”qmake inetNoTcp.pro”. Then change working path to *$OTWLAN/otwlan/otcp* and do the usual “qmake;make clean;make”.

Step 3 *Build bin/otwlan*
Do a “cd $OTWLAN/otwlan/src” and execute the sequence “qmake **“CONFIG+=usetcp” otwlan.pro**; make clean; make”.
Step 4, *5, 6*
Identical to the steps 4 to 6 in section 11.1.

11.3 Running the First Time

oTWLAN remembers its setting across sessions and this information in stored in a file. This function uses the Qt class *QSettings* which stores information under “$HOME/.config” on Linux. We have assigned the organisation name *gosikt* and the application name *otwlan*. The absolute file name then becomes *$HOME/.config/gosikt/otwlan.conf*.

The first time the program starts this file is missing and you are requested to give some addition information. Set the project directory to *$OTWLAN/otwlan* and the simulation directory to *$OTWLAN/otwlan/examples/myfirstrun.* The latter is a simple two node network that should run immediately without any additional configurations. Use the *oTWLAN* menu “Setting->Show” to inspect your configuration variables.

Figure 11.1 shows the variables specified by the *otwlan.conf* file. The *[project].home* must point to the *$OTWLAN/otwlan* because the program must have access to the files in the directory *$OTWLAN/otwlan/nedFiles*. The *simulationHomeDirectory* should point to a valid simulator input data set as explained in chapter 5. However, wrong setting is not fatal. The program starts with an empty playground. Use “Project->Open” to open a valid data set.

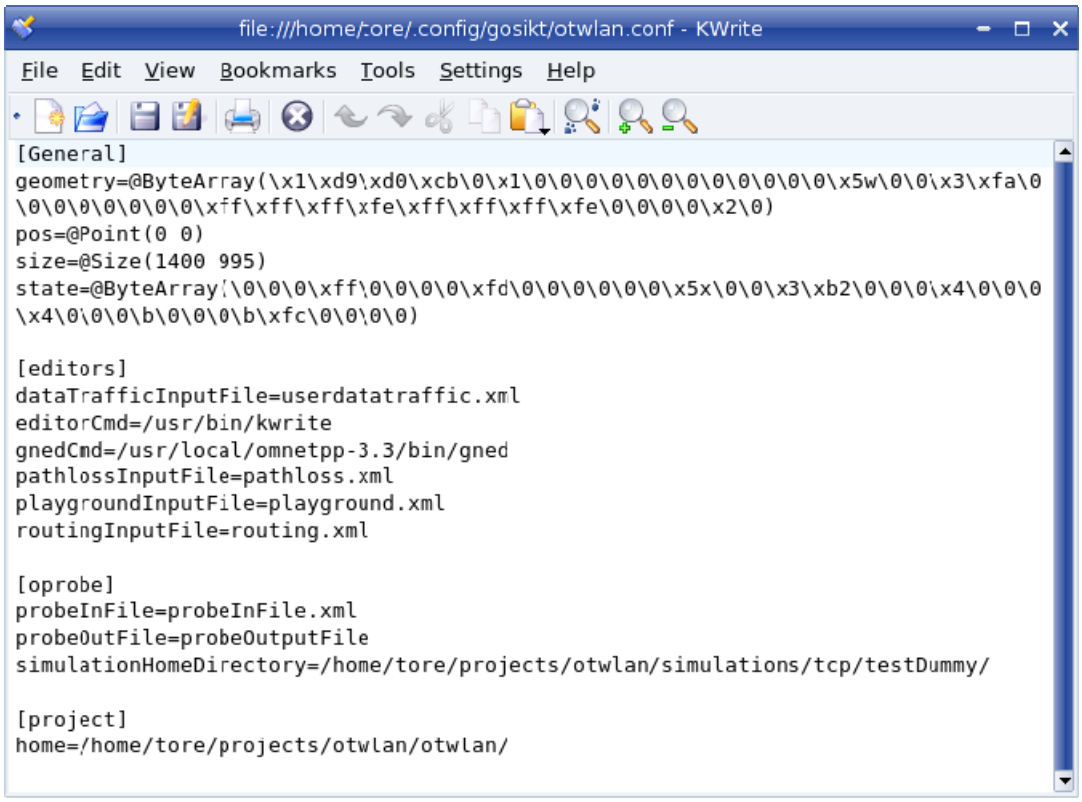


Figure 11.1 The content of the otwlan.conf. A copy is supplied under $OTWLAN/otwlan.

12 Conclusions and Remarks

The *oTWLAN* software is developed under the FFI-project Fundamental Technologies and Trends in Information Security (GOSIKT) and is the first step towards developing a simulator that shall model NATO PKI in the tactical domain where the network nodes must rely on radio based communication with limited transmission capacity.

Model validation. Much effort has been made to validate the simulator. Chapter 9 simulated some scenarios where analytical expressions are available. Here we compared simulated results and theoretical results. As we progress towards multihop networks in chapter 10, analytical results become unavailable and the strategy was to gradually increase the complexity with the objective to detect abnormalities.

Scenario initialization. A common pitfall is not to initialise the simulation scenario correctly. *oTWLAN* provides a set of graphical editors by which the user configure the simulator's input data. These editors make some consistency checks and produce XML files that are processed by simulator kernel upon start. Practical experience has shown that the run-time checks done in the kernel catches most of the inconsistencies in the input data. However, they are trapped by assert()-statement. This may be inconvenient for inexperienced users but was a shortcut we had to take due to lack of programming resources. Upon termination, *oTWLAN* produces a file containing a set of counters, which is sampled data per node basis. Counters showed to be an

effective tool to detect erroneous setting of the input data. For example, if node N0, say, detects many CRC errors despite the fact it should not, this may indicate inappropriate setting of the power level in the network.

Output data analysis. Many simulators have no tools to do statistical analyses and some studies have pointed out that this is a practical problem [4,24]. By means of the open source project *oProbe* [2], *oTWLAN* inserts probe objects in the program code. These probes collect samples and perform data analysis of steady-state performance at run-time. The simulation process terminates automatically when the desired accuracy is reached. *oTWLAN* is an infinite horizon simulator, that is, we suppose as the simulator runs longer and longer, a limiting steady-state distribution independent of the initial state exists. If the transient period is set too short, biased results are produced. However, the statistical analysis package implemented tests the sampled data for correlation and automatically prolong the run length until a predefined upper limit is reached. This technique solves the problem with correlated samples but also makes the simulator less sensitive to short setting of the transient period.

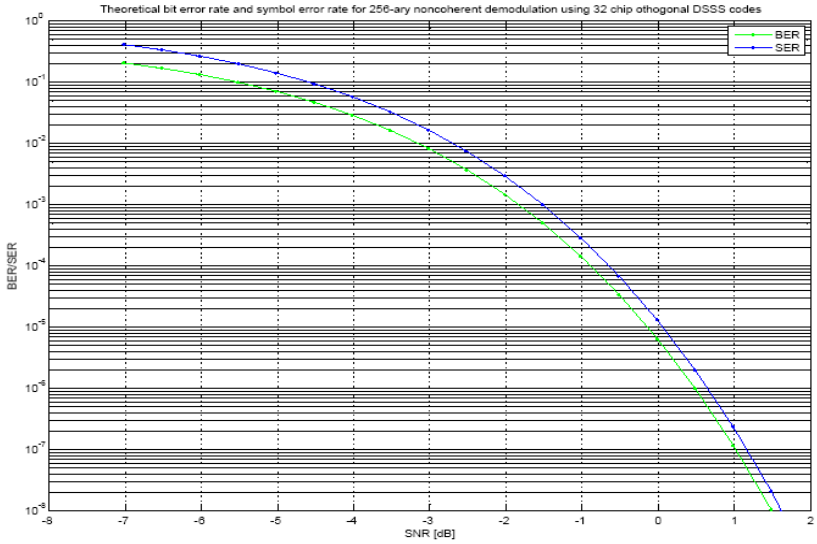
Software quality assurance. The source code has been compiled with "warn all" enabled and all warnings have been corrected. Memory leaks and use of un-initialised variables have been detected by means of *valgrind*. We have extensively used the *assert()*-statement in our code to detect fatal software errors as early as possible since this disburdens the debugging process. It also prevents producing simulation results from inconsistent or erroneous input data. When simulating a complex scenario, it is often impossible to detect errors in the simulator's input data from the output data.

Object-oriented modelling. RF communication is modelled according to the real world where a wave propagates through air and experience a pathloss depending on the distance. All incoming RF waves to a receiver go through comprehensive analysis where interference, background noise and the air frame section considered determine the next receiver state. The benefit of this approach is a good emulation of a real scenario. The drawback is a higher computational cost and networks larger than 200 nodes may run slowly depending on the connectivity.

Why not NS-2? The most used simulator in MANET research is NS-2 [27]. Some of the project team members have experience with the NS-2 and found it more attractive to develop a new simulator rather than modifying the NS-2 software. The focus for our research is MANETs for the military tactical domain where both the protocols and the radio solutions differ from what is modelled in NS-2. NS-2 also has some shortcomings with regard to structure. For example, NS-2 has no module concept as *OMNeT++*. The concept of modules appeared to be an efficient tool for structuring the simulator.

Appendix A Symbol Error Rate

All the three *oTWLAN* radio versions use the same symbol error rate (SER) table



Demodulation:	-5	-4	-3	-2	-1	0	1	2	3	4
F1:	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2

Figure A.1 SER versus SNR for demodulation (CRC16, LI and PHY payload) and F1.

Appendix B Egli Single-hop Network

This appendix continues the analysis of the scenario specified in section 10.4. We modify the traffic generators to restrict the traffic streams to neighbours only (single hop network). This is done by activating the optional user facility "to all RF neighbours using link cost limit" on the traffic pattern page in the traffic editor. Here we may also apply a link cost condition to control whether a link shall receive traffic or not. All RF connected neighbours of the originator receive packets when the link cost limit is set to 1.49. Only high quality neighbours receive packets if the link cost limit is set to 1.00. All transmitters send at -6.5dBm while the other node parameters remain at the same values as earlier.

From figure B.1 we can see a sudden drop in performance when the link cost limit increases from 1.3 to 1.4. More low quality links are taken into use and the LLC retransmission rate increases. Figure B.2 illustrates a high retransmission rate even at low load level when the low quality links carry traffic.

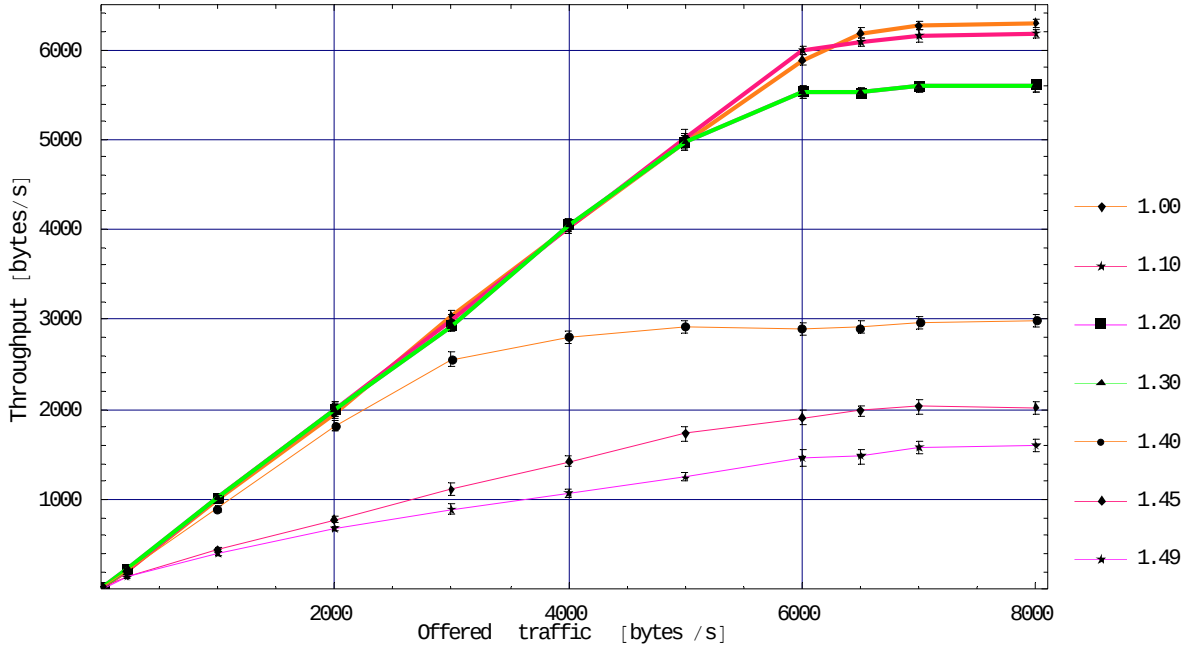


Figure B.1 Throughput versus offered traffic. The legend expresses the link cost limit applied. The plot for link cost 1.20 and 1.30 are identical.

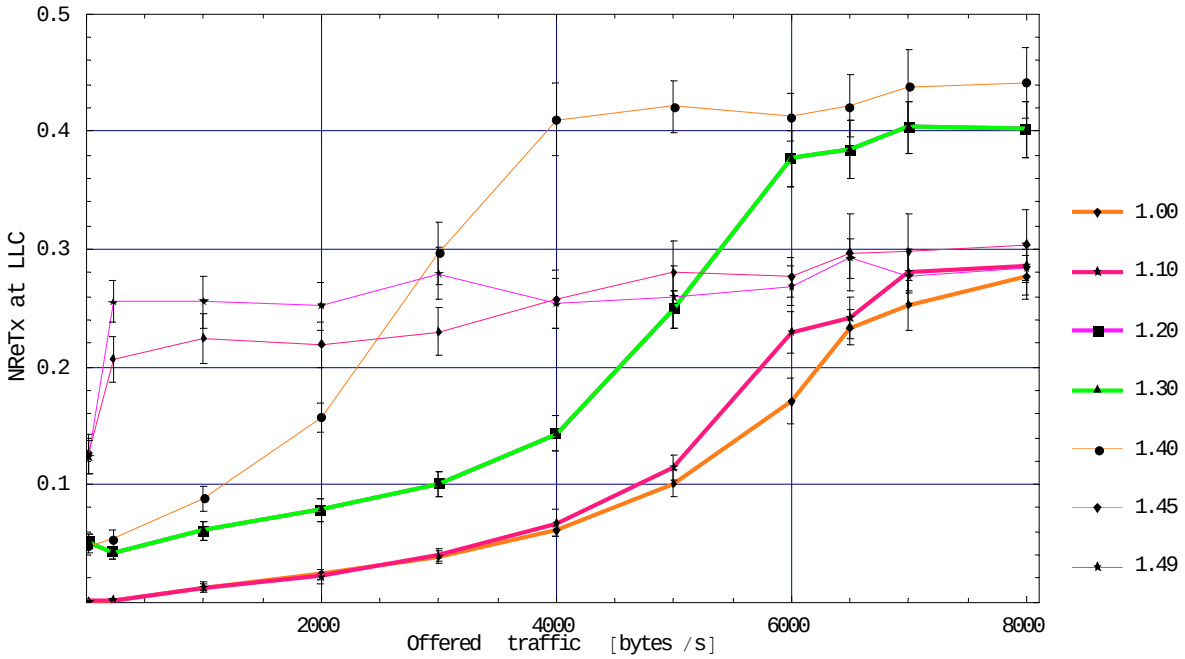


Figure B.2 Average number of retransmissions at the LLC layer retransmission versus offered traffic. The legend expresses the link cost limit applied. The plot for link cost 1.20 and 1.30 are identical.

Acronyms

AHA	All hearing all (complete network topology)
ARQ	Automatic Repeat reQuest
CAS	Carrier Sense
CL	ConnectionLess
CNR	Combat Net Radio
CO	Connection Oriented
DARPA	Defence Advanced Research Projects Agency
DOM	Document Object Model
DSSS	Direct Sequence Spread Spectrum
DT-PDU	Data Protocol Data Unit
GUI	Graphical User Interface
GUIA	GUI Automatic
ICI	Interface Control Information
IP	Internet Protocol
IP-SAP	Internet Protocol SAP
ISO	International Organisation for Standardisation
LLC	Logical Link Control
MAC	Medium Access Control
MAC-E	MAC Entity
MAC-SP	MAC Service Provider
MANET	Mobile Ad-hoc NETwork
MLPP	Multi-Level Precedence and Preemption
NIC	Network Interface Card
NM-SAP	Network Management SAP
OS	Operating System
OSI	Open System Interconnection
PCI	Protocol Control Information
PDP	Packet Data Protocol
PDU	Protocol Data Unit
PTT	Push To Talk
RF	Radio Frequency
SAP	Service Access Point
SDU	Service Data Unit
SQL	Structured Query Language
TCP	Transport Control Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UDP	User Datagram Protocol
UE	User Environment or User Equipment
UI	User Interface
UTL	Utility

WAN	Wide Area Network
XML	Extensible Mark-up Language
xxx-E	xxx Entity (e.g., LLC-E)
xxx-SAP	xxx Service Access Point (e.g., LLC-SAP)
xxx-SP	xxx Service Provider (e.g. MAC-SP)

References

[1] The Discrete Event Simulation OMNeT++, www.omnetpp.org

[2] Tore J Berg, ”oProbe - an OMNeT++ Extension Module”, http://soureforge.net/projects/oprobe

[3] Bjørnar Libæk, “A TCP module for the oTWLAN”, in preparation.

[4] Kurkowski, Camp and Colagrosso, ”MANET Simulation Studies: The Incredibles”, ACM's Mobile Computing and Communications Review, vol. 9, no. 4, pp. 50-61, October 2005.

[5] Berg, et.al, ”Spread spectrum in mobile communication”, The Institution of Electrical Engineers, 1998, ISBN 0-85296-935-X

[6] Alan Ezust and Paul Ezust “An introduction to design patterns in C++ with Qt 4”, Prentice Hall 2007, ISBN 0-13-187905-7

[7] Qt Centre, www.qtcentre.org

[8] John Jubin and Janet D. Tornow, “The DARPA Packet Radio Network Protocols”, Proceedings of the IEEE, January 1987.

[9] Edward N Singer, “Land Mobile Radio Systems”, Second Edition, Prentice Hall 1994

[10] Ramakrishna Gummadi, et.al., “Understanding and Mitigating the Impact of RF Interference on 802.11 Networks”, 2007, www.seatle.intel-research.net/Publication/2007

[11] Daniel Willkomm and Marc Loebbers, “A Mobility Framework for OMNeT++ User Manual”, January 12th, 2007.

[12] I. Ramachandran and S. Roy, “On the Impact of Clear Channel Assessment on MAC Performance”, Global Telecommunications Conference, 2006.

[13] Marc Lobbers and Daniel Willkomm, “A Mobility Framework for OMNeT++”, January 12th, 2007, www.omnetpp.org.

[14] Open system interconnection - Basic reference model, ISO/IEC 7498.

[15] Conventions for the definition of OSI Services, ITU-T Recommendation X.210.

[16] Krzysztof Pawlikowski, “Steady-State Simulation of Queueing Processes: A Survey of Problems and Solutions”, ACM Computing Surveys, Vol. 22, No.2, June 1990.

[17] R. G. Cole and B. S. Farroha, “Implications of Precedence and Preemption Requirements on Packet Based Transport Architectures”, MILCOM 2007.

[18] B. S. Farroha et.al, “Requirements and Architectural Analysis for Precedence Capabilities in the Global Information Grid”, MILCOM 2006.

[19] William Stallings, “Data and computer communications”, Macmillan Publishing Company, 1985.

[20] Mathematica, www.wolfram.com

[21] John G. Proakis, “Digital Communications”, McGraw-Hill international editions, Third Edition 1995

[22] Thomas Dreibolz, et.al. , “SimProcTC - The Design and Realization of a Powerful Tool-Chain for OMNeT++ Simulations”, 2nd International Workshop on OMNeT++, March 6th, 2009.

[23] Frank Y. Li, et.al. , “Does Higher Datarate Perform Better in IEEE 802.11-based Multihop Ad Hoc Networks?”, Journal of communications and networks, 2007, Vol 3.

[24] Kurkowski, Camp and Colagrosso, ”MANET Simulation Studies: The Current State and New Simulation Tools”, Technical Report MCS-05-02, The Colorado School of Mines, February 2005, http://toilers.mines.edu.

[25] NATO Consultation, Command and Control (C3) Board, “NATO Public Key Infrastructure (NPKI) Certificate Policy”, AC/322-D(2004)0024-REV2, 2008.

[26] NATO Consultation, Command and Control (C3) Board, “Statement of Technical Characteristics for the NATO Public Key Infrastructure, AC/322-N(2008)0004, 2008.

[27] The NS-2 simulator, http://nsnam.isi.edu.nsnam/index.php.