Media Model for the Emulation of Wireless Broadcast Networks

Bachelor Thesis

by

Matthias Jansen

from

Wuppertal

submitted to

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

January 2007

Advisors:
Dipl. Inf. Björn Scheuermann
Dipl. Wirtsch.-Inf. Christian Lochert

Acknowledgments

A lot of people helped me during my work on this thesis to whom I wish to express my gratitude. At most I have to thank my advisors Björn Scheuermann and Christian Lochert for the great support during the implementation of the emulator and even greater support during writing this thesis. I also must thank Dr. Klaus Hinrichs for his view from an "outsider" to this thesis. He helped me a lot to find lacks of explanation and was a great source of transliteration which demonstrable improved my English phrases. I want to thank our cat, too, for running over my keyboard and destroying complete sentences which forced me to think the paragraph over again. This might have been constructive one or two times and I must thank "die Ärzte" for their great music which kept me up on late night sessions to complete the paragraph again. At last, but not least, I want to thank my wife Daniela for being so amicable even in times when she saw nothing but my back for days.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

These days we cannot imagine a life without communication and even wireless communication has already entered many aspects of our life. Today there exist already many more cellphones than conventional telephones and the gap is still growing. With more and more upcoming DSL connections the number of IEEE802.11 WLAN access points and clients increases, too. But wireless communication is not limited to those infrastructure driven networks. Wireless nodes could build a mobile ad-hoc network which means that all participating nodes act on the same level. Each node in a mobile ad-hoc network is router and client at the same time. The possibilities a mobile ad-hoc network could offer could easily overcome those of any infrastructure network. For example one very exciting field of research is the use of car-to-car communication. Cars should build networks only by themselves to exchange messages about traffic jams, accidents or even latest news. Compared to infrastructure networks communication in mobile ad-hoc networks are much more complex. Due to node movement connections can easily break and usually a packet needs to be forwarded by several other nodes in order to arrive at the correct destination. These and other problems push the development of different techniques and protocols to keep this sometimes very chaotic communication alive.

Developing new wireless communication protocols is often hard work. Even if it happens to be that there exists already usable hardware, in most cases testing with real hardware is difficult because of several reasons. One reason certainly are the high costs for all the mobile nodes needed for a good testing environment. Another reason is the difficult and time-consuming running of different testing scenarios. Despite of all these drawbacks real testing with real hardware offers the most realistic results and should be done at the

end of the development to verify all results which have been generated with simulators or emulators.

So the first approach for testing is often the use of a common network simulator like the well-known ns-2 [NS2]. A network simulator is an application which simulates a completely virtual environment. It includes the network and physical layer, the operating system layer as well as the protocol implementation and the testing tools. These simulators provide all functions needed to rebuild a complete wireless scenario including a wireless propagation model, a huge number of mobile nodes, collision simulation and many more features to be as realistic as possible. It is usually a good idea to use a simulator to get a first set of results. But despite all possibilities a simulator can offer it differs from reality in one important fact: a simulation is neither realtime nor a real environment so the results must be handled with a certain care. The environment of a simulator is artificial. There is no real operating system involved which has a certain impact on the overall performance. Also the given hardware might act differently compared to the simulated system. It is impossible to include all parameters which influence a real testing environment. Contrary to the use of a simulator a network emulator only imitates a certain part of the environment, in most cases the physical layer, and available software is able to work without noticing that there is, e.g., no real wireless network available. This allows the use of already existing software with the emulator. The use of real software implies in most cases that the emulator must not run much slower than real time because this would break, e.g., a client server communication which uses timeouts and raising these timeout values would make communication nearly impossible. That is the reason why a network emulator is in some cases less realistic compared to a simulator and usually supports a much smaller number of mobile nodes. As the simulator can stretch time as needed it can perform much better calculation and doesn't need to complete calculation in a certain time. So the simulator can easily calculate several minutes for just one simulated second. A quite different situation is the impact of the used operating systems and other system environment details where the emulator beats the simulator in realism. A benefit of the emulator is the possibility to reuse the source code in a real implementation with very few changes. The ability to use current applications for testing like a specific routing daemon might shorten preparation time considerably.

There exist already many emulators for wireless networks but an emulator capable of running a realistic radio propagation model like a simulator in real time with support for new and non-existing hardware interfaces is not available to the best of our knowl-

edge. One reason for implementing a completely new emulator was the intention to test the CXCC protocol [SLM07] developed at this chair by Björn Scheuermann and Christian Lochert. This protocol needs special hardware hooks which are not provided by any hardware yet. Tests have already been done using a simulator and a sensor network but to verify the results with accordance to implementation details and impacts of the underlying operating system an emulator is needed which supports the emulation of currently unavailable hardware in cooperation with a realistic radio propagation model. These challenging requirements make high demands on the features of the emulator like a fine-grain timer. In the context of this thesis we present an emulator that supports a timer granularity below 100 μs. In order to emulate a 10 MBit wireless network the emulator has to be able to switch from transmission start to packet reception for a small packet (100 byte) in about 100 byte / 10 Mbit/s = 76,23 μs and has to support a realistic radio propagation model including collisions and transmission delay. Every node has a set of states which are switched within the mentioned time and keep information about all network settings. This information is needed to decide whether the communication could be completed successfully or not. To be able to emulate a real IEEE802.11 WLAN even smaller timing intervals are needed because of the tiny backoff times. In order to be able to run on just one physical machine to be most cost-effective at the same time some restrictions for the design had to be observed.

This thesis describes the Low Latency Wireless Emulator (LoLaWe) which is features a fine-grained timer thread with timer intervals of about 70 μs. This enables the emulator to simulate a complex radio propagation model in realtime. Further dynamic loading of different emulation engines is supported for easier development. LoLaWe is able to run a complex radio propagation model with a network bandwidth of 1 MBit and 20 nodes on a single computer.

The remainder of this thesis is structured as follows. The next chapter will deal with other approaches to emulate or simulate wireless networks and show the differences to the newly presented emulator. The third chapter will describe a common radio propagation model to better understand the special aspects in Chapter 4 which deals with the main aspects of implementation of the new emulator. Chapter 5 will present a performance analysis where the new emulator is compared to the MarNET emulator. Finally, Chapter 6 concludes the thesis.

Chapter 2

Related Work

2.1 Simulators

Simulators provide many functionalities and it is common practice to evaluate a newly developed protocol using an implementation in a simulator first. It is possible to simulate a large number of nodes with a realistic radio propagation model depending on the testing machine and its hardware specifications like CPU and RAM which can't be handled by any emulator. This allows the use of a much more complex model compared to an emulator running in real time and so the results could provide a better representation of the physical behavior of radio propagation. The major drawback of using a simulator is the complete abstraction from real implementation and running systems. That means that the results have to be treated carefully because the results from the radio model may be realistic but the rest of the system might act different compared to a real physical node with a wide range of possible influences to the network connectivity (operating system, hardware interoperation, driver implementation, other running software, etc.). It is also necessary to implement the protocol differently compared to a real implementation. This means that the protocol needs to be implemented again in order to run it on real hardware. Contrary to this a protocol implementation used in the emulator described here could be transfered to real and identical hardware and be used immediately. As the number of simulators is very large and the functionality does not much differ no special simulator is described here but the network simulator ns-2 [NS2] as the probably most commonly used network simulator shall be named as an example.

2.2 Emulators

Contrary to simulators emulators only provide a certain piece of the overall environment. There exist lots of different emulators with different design goals. These goals all need different degrees of emulation. This could vary from just the network layer to a complete emulated computer or every stage in between. There are emulators for special communications like the Ohio Network Emulator ONE [AO96] which is specialised for satellite communication or emulators which only control the traffic going through or into a router like the NIST Net [CS03]. Other emulators focus on visualisation (JEmu [FTO01]) and less on a realistic radio model or distribute the emulation on multiple machines as it is done in MobiNet [MRBV04]. Some emulators use virtualization as a presentation for virtual nodes like MarNET [SHF05] and [BSR+05] which is also proposed in [MHR05]. Another approach is the use of special hardware to emulate radio propagation which is used in [JS]. Obviously this isn't very portable and is another source of costs. There also exist emulators with a limited purpose like [WLZ+04] and MEADOWS [LNH+04] which aim at the emulation of sensor networks. There are many other emulators but to better understand some basic techniques two emulators are representatively described: MarNET and NEMAN.

2.2.1 MarNET

MarNET is an emulator based on the paravirtualization [PAR] provided by XEN [XEN]. XEN is a so called hypervisor which enables a PC to run multiple instances of operating systems at the same time and allows the user to switch between them. The operating systems are so called paravirtualized which means that they need modifications to be able to run with a hypervisor. Each virtual mobile node is represented by a virtual instance of Linux (DomU) running under the XEN hypervisor. Figure 2.1 shows the main organization. The main advantage is the complete separation of each node which allows a very easy way to implement protocols and applications without special precaution because the node itself doesn't know that it is only a virtual instance and behaves like a real physical node. Only the network communication between the nodes is simulated and controlled by an application running on the main Linux instance (Dom0). This application has an easy to use graphical interface which allows to start a number of virtual

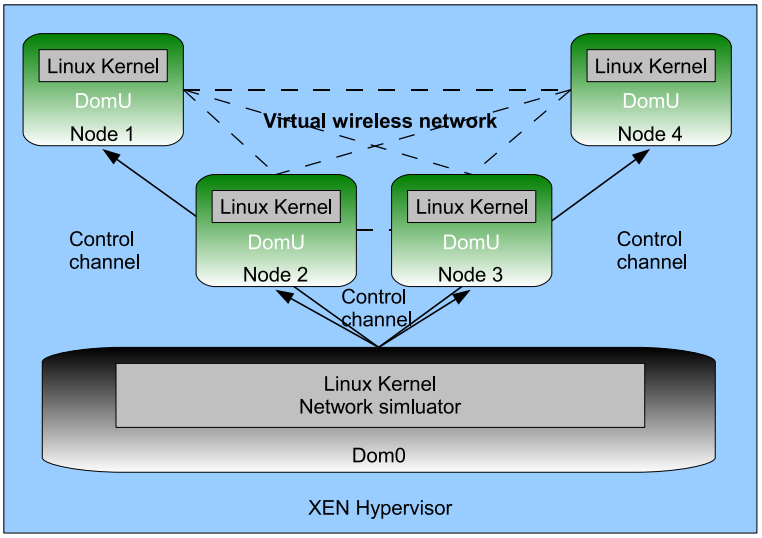


Figure 2.1: Design of a paravirtualized emulator, here using XEN

nodes. The simulation data can be provided with a ns-2 compatible scenario file. The complete emulation can be followed using a graphical representation of all virtual nodes where the edges from node to node show the connection quality. But the emulator allows only to set the bandwidth, a transmission delay and a ratio for packet delivery errors so that the connection quality is only calculated using the pure distance. Unfortunately the main advantage is also the main drawback because switching between complete Linux instances is obviously very expensive so that the latency for sending packets from node to node could easily reach about 1 to 2 ms with peaks of about 15 ms. As already discussed a complex emulation would need timer granularity of only a few microseconds. That is the reason why it is quite difficult if not impossible to rebuild complex radio propagation models using a paravirtualization because of the huge overhead. As a consequence MarNET is unable to provide the required features. In other cases where low latency isn't needed this approach nevertheless provides a very easy and intuitive interface for emulation of completely separated virtual nodes.

2.2.2 NEMAN

A Network Emulator for Mobile Ad-Hoc Networks (NEMAN) [PP05] was developed at the University of Oslo derived from MobiEmu [ZL02]. Like MobiEmu NEMAN has a graphical frontend to the emulation settings, node positions etc., and controls the under-

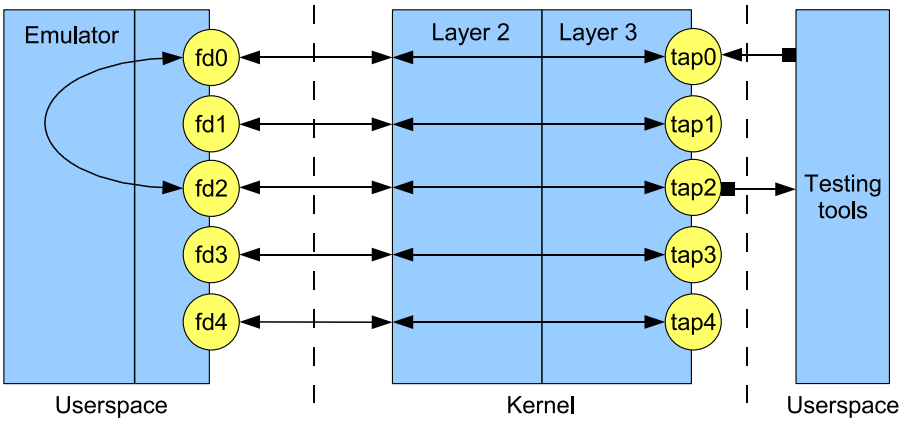


Figure 2.2: Design of an emulator using TUN/TAP devices as representation for each mobile node

lying topology manager which is the real emulator. It uses TUN/TAP devices, virtual ethernet interfaces provided by the Linux kernel which connect to a running userspace process instead of a real physical layer on OSI layer 2, as a representative for the mobile nodes (see Figure 2.2). NEMAN opens one TAP device for each node and connects them with the topology manager which is controlled by the graphical user interface connected with the tap0 device and a separated control channel. In contrast to the MarNET emulator this emulator only runs one instance of a Linux kernel without the need for the expensive switch between different kernel instances. But this means that all test applications run on the same systems and have to share the same network stack. To work correctly these applications need special treatment of socket connections which might disqualify some applications but the advantage of the better timing latency should outweigh the drawback of special application requirements. The here presented emulator uses also TAP devices and thus has the same benefits and disadvantages. However, the main difference is the fact that NEMAN doesn't support a realistic radio propagation model which would include collisions, noise-signal-ratios, delivery latency etc. To be able to emulate these functions at least a high precession timer is needed which is not included in NEMAN.

Chapter 3

Radio Propagation Model

Wireless networks are different in many aspects compared to a cable network. The media (air) is shared by all actively participating nodes and in most cases each node transmits the signal in every direction. Each node is in one of the 4 possible states: idle, receiving (rx), transmitting (tx) or blocked. Normally a node is in the idle state. While being in the state receiving the node only changes to the state blocked after an interruption while receiving. In principle all situations where the signal level is not strong enough are called collisions.

Sending in a 360° radius leads to the possibility that not only the target node receives the signal but also every other node within the maximum range of transmission. This signal, which a node receives without being the addressed one, increases the noise level. The basic noise is an overall hissing due to all other radio communications, electronic devices like microwaves and other sources of radio waves. With increasing network density the noise level at each node increases due to additional noise of multiple nodes in the neighborhood. When a node receives a packet with itself as destination the signal power of this specific packet needs to be significantly higher as the current noise level or else it is impossible for the node to receive and decode the data correctly. If it is impossible to decode a packet or the node receives packets from two different other nodes the failure is a collision. This can happen when a nearby node sends data to a third node while the distance to the current nodes communication partner is bigger because signal power decreases as distances increases. In this case the signal of the interrupting node is stronger. Figure 3.1 shows a very simple example where two nodes are communicating (node 1 and node 2) and are interrupted by node 3 because the signal

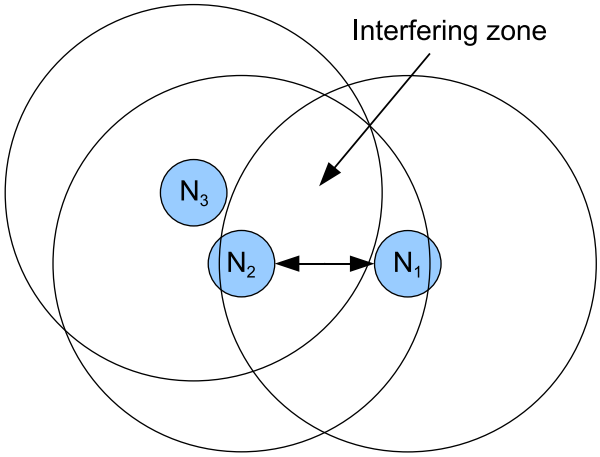


Figure 3.1: Node 3 could interrupt the communication between node 1 and 2

of node 3 is stronger than the signal of node 1 so node 2 is unable to decode the data from node 1 correctly. After the sending interruption node 2 stays in collision or blocked state until node 1 finishes the transmission because node 1 is unable to detect the collision while transmitting a single packet. Figure 3.2 shows a state diagram with all possible transitions. In Figure 3.3 node 2 sends a packet to node 1 after 2 ms with a signal power of -60 dBm. Node 4 starts a transmission to another node after 4 ms but the signal power isn't high enough to interfere the communication between node 1 and node 2. But after 5 ms node 3 starts transmitting packets with a signal power high enough to raise the noise at node 1 above the threshold and the signal from node 2 is disturbed. As node 2 can't recognize the interruption it finalize the transmission despite the fact the node 1 could not decode the complete packet and so rejects the rest of the transmission.

To be able to emulate a virtual node to node communication a radio propagation model must be used which defines how the wireless communication works in detail. This can range from a simple distance boundary system which only checks if the target node is within a certain radius from the source node to a complex simulation system which calculates the sending of packets through air including physical aspects like reflection. For a great flexibility and to be able to use different radio propagation models the emulator loads the emulation engine, which defines the radio propagation model, dynamically during the start defined by a command line parameter. This allows the users to create many emulation engines that fit best with their requirements without the need to implement the details directly in the emulator. These engines can easily be distributed amongst researchers for better testing and comparison of results.

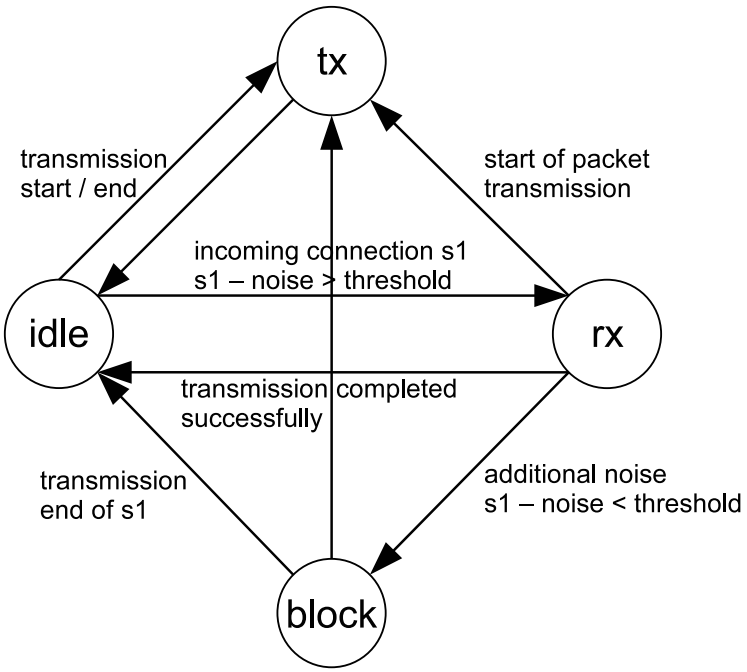


Figure 3.2: Different node states

As seen before the distance between two wireless mobile nodes is of great importance because the distance has the greatest influence to the signal power. In order to calculate the power at the receiving node several things have to be taken into account. These include the sending power, the antenna gain and even more important the power loss along the distance between sender and receiver. To calculate the power loss the three most popular models are "Free Space Loss", "Two-ray Ground Reflection Model" and "Shadowing Model" which are all implemented in ns-2 [RAD].

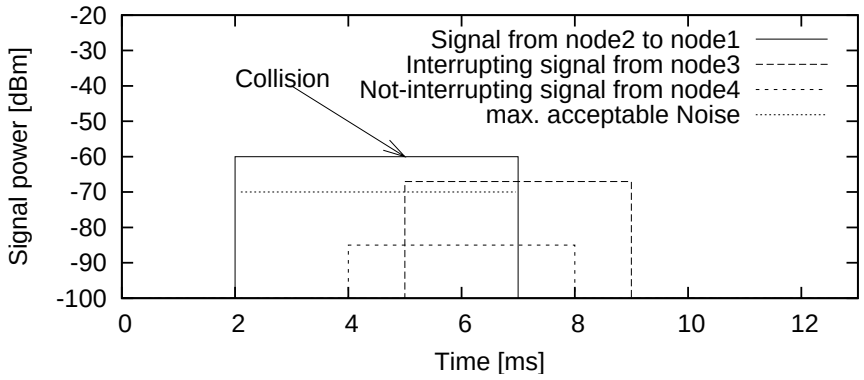


Figure 3.3: Communication interruption during transmission

3.1 Free-Space Loss Model

The free space loss is the simplest model which assumes that there are ideal propagation conditions. This means that there is a clear line-of-sight path between sender and receiver and the signal runs only on one straight line. H.T. Friis developed the following formula [Fri46] to calculate the signal power $P_r(d)$ at the receiver at distance d

$$P_r(d) = \frac{P_t G_t G_r \lambda^2}{(4\pi)^2 d^2 L} \tag{3.1}$$

where P_t is the sending power, G_t and G_r are the antenna gains of sender and receiver, L is the system loss and λ is the wavelength. Normally ns-2 defines $G_t = G_r = L = 1$. A convenient way to express the free space loss is in terms of dB as shown in (3.2)

$$FSL_{dB}(d) = 20(\log_{10}(d) + \log_{10}(f)) + K \tag{3.2}$$

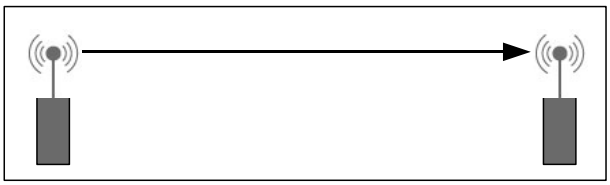
where d is the distance, f is the frequency and K is a correction constant depending on the units used for d and f. In the specific case of this newly designed emulator where a 2.4 GHz WLAN network is emulated the formula simplify to

$$FSL_{dB}(d) = -40.4 - 20\log_{10}(d) \tag{3.3}$$

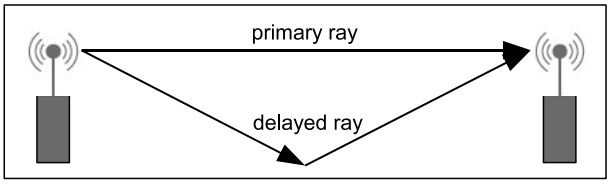
The Free-Space Loss formula is good for near communication but is increasingly inaccurate for far distances.

3.2 Two-Ray Ground Reflection Model

The Two-Ray Ground Reflection Model is an improved version of the Free-Space Loss model. In reality a single straight line of communication is seldom the only signal propagation. The Tow-Ray Ground Reflection Model also includes signals which arrive at



(a) Free-Space Loss



(b) Two-Ray Ground

Figure 3.4: a shows the simplified propagation used in the Free-Space Loss Model, b shows the propagation used by the Two-Ray Ground Reflection Model

the receiver due to reflections from the ground (see Figure 3.4). Therefor (3.4) includes the heights of the receiver and sender (h_r and h_t).

$$P_r(d) = \frac{P_t G_t G_r h_t^2 h_r^2}{d^4 L} \tag{3.4}$$

This formula decreases faster than (3.1) as the distance increases but the Two-Ray Ground Reflection Model is inaccurate for short distances because of the oscillation caused by the combination of the destructive and constructive rays. That is the reason why some simulators (and this emulator) uses a combination of Two-Ray Ground and Free Space Loss models chosen by a threshold d_c where both models produce the same value.

$$d_c = \frac{(4\pi h_t h_r)}{\lambda} \tag{3.5}$$

3.3 Shadowing Model

Both presented models, Free Space Loss and Two Ray Ground, are deterministic assuming a perfect circle around the sending node. In reality the signal is reflected many times so the signal power contains a certain random factor also known as fading effects. The

Environment	β	σ_{dB} **of** X_{dB}
Outdoor urban area	2.7 to 5	4 to 12
Outdoor free space	2	4 to 12
Indoor line-of-sight	1.6 to 1.8	3 to 6
Indoor obstructed	4 to 6	6.8

Table 3.1: Parameter examples for the Shadowing model

shadowing model tries to include these effects with a certain random factor. This factor needs to be configured with a few parameters to match the environment to be simulated because an open terrain is much different compared to a terrain in the city with multiple reflection points. The complete equation, containing two parts of the model – the path loss model and the shadowing model – is shown in (3.6) where β is the pass loss exponent and X_{dB} is a random value of a Gaussian distribution with $\mu = 0$ and σ as a parameter. Both β and σ need to be specified. Table 3.1 lists typical values for these parameters which are defined in [RAD].

$$\left[\frac{P_r(d)}{P_r(d_0)}\right]_{dB} = -10\beta \log\left(\frac{d}{d_0}\right) + X_{dB} \tag{3.6}$$

3.4 Emulation engine

The advantage of the provided emulation engine is the use of a complex radio propagation model. This model includes calculating the signal power of a transmission at the receiver and simulates collisions. To better represent the conditions each node is in one of a set of states (see Figure 3.2). The Free-Space Loss and the Two-Ray Ground model are implemented. Figure 3.5 compares the Free-Space Loss, the Two-Ray Ground and the Shadowing model. The parameters are 20 dBm (equivalent to 100 mW) sending power, a wavelength of 0.125 meter for a 2.4 Ghz WLAN and a sender and receiver height of 1 meter. The additional parameters for the Shadowing model are set to $\beta = 1.2$ ($\beta = 1.5$), $d_0 = 1$, the standard deviation for the X_{dB} distribution $\sigma_{dB} = 5$ and mean $\mu_{dB} = 0$.

It is easy to see at Figure 3.5 that the Two-Ray Ground Model is unrealistically high at short distances, which is the reason for not using only the Two-Ray Ground model in simulations, but is decreasing significantly faster than the Free-Space Loss model. The

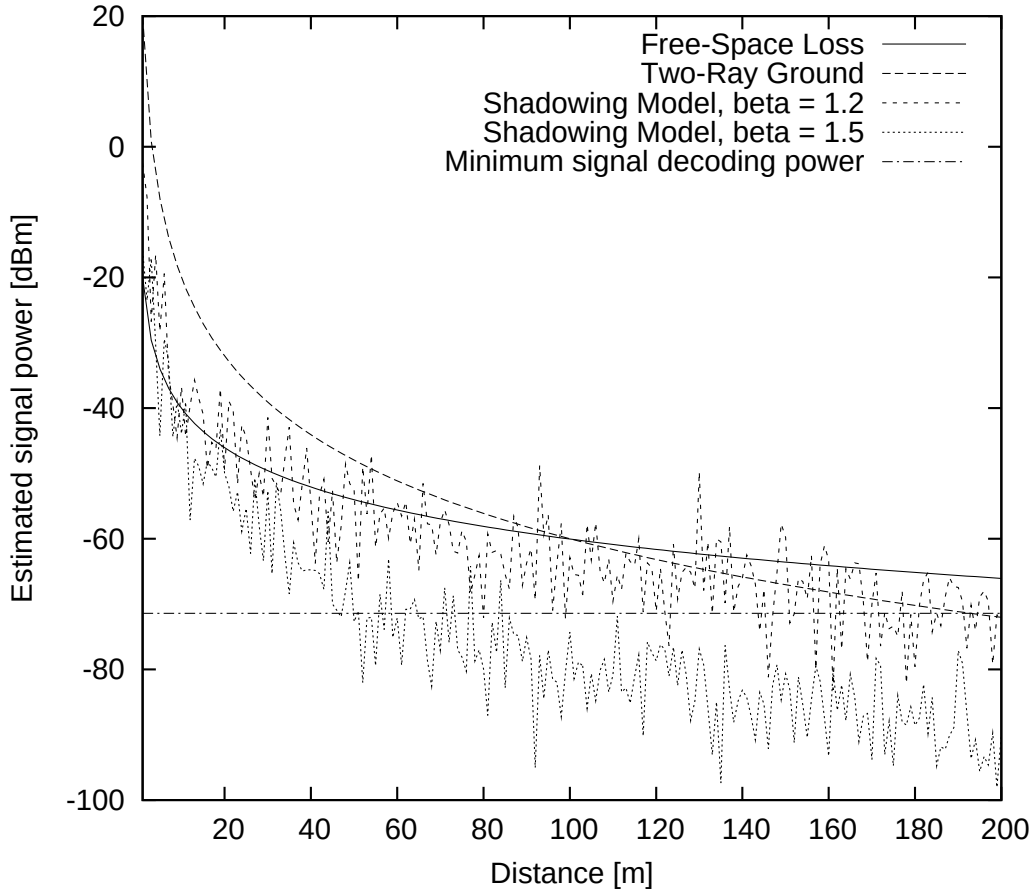


Figure 3.5: Comparison of Free-Space Loss, Two-Ray Ground and Shadowing Model

d_c from (3.5) is about 100 meter where both models provide the same estimated signal power. The Shadowing model is non-deterministic which might be more realistic but highly depends on the parameters which need a lot of testing to get the right values for the right environment. As shown in Figure 3.5 the slight adjustment of β with a value 0.3 leads to a significantly different transmission power so the results of the Shadowing Model need to be treated with a certain care.

Chapter 4

The Emulator "LoLaWe"

LoLaWe has been developed as an emulator for wireless broadcast networks. As being a wireless network emulator it abstracts the wireless network and enables the use of multiple virtual mobile nodes communicating with each other through a virtual media on a single computer. To be able to compare the results achieved with an emulator it has to be as close as possible to real testing scenarios. Therefore the emulator needs to know the boundaries of the testing environment which includes the number of nodes, their movements and the size of the testing area where the movements of the nodes take place. The testing area is assumed to be rectangular with a defined width and height. Every node has an initial position and a number of subsequent waypoints associated with a corresponding time coordinates. Using those information and the simplifications that every node moves on a straight line with constant speed between two positions and all nodes are on a plain 2D territory the current position at any given time can be calculated. Figure 4.1 gives an example of a visualized virtual scenario. All information needed by the emulator are provided by the user through a so called scenario file. In Appendix B a complete description of the file format is provided.

After loading the scenario file the emulator opens a number of TUN/TAP network interfaces (tap0 to tapX, where X is the number of nodes in the scenario), each tap device representing a single virtual node. When all interfaces are open the emulator starts the real emulation. The emulator waits for a packet sent through any of the open tap devices. When the emulator receives a packet through one tap device the packet is examined and the target node is extracted. Now the actual positions of both the sending and receiving node are calculated in order to determine the signal power and to decide whether the

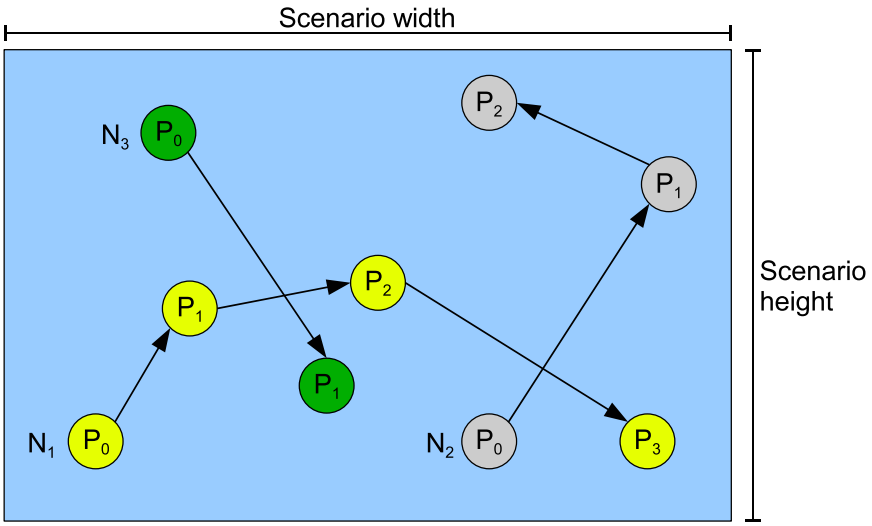


Figure 4.1: Example of a virtual scenario. N_x are virtual nodes and P_x are positions

packet could be transmitted correctly or the distance between the two nodes is too long. If the target node is within the sending radius of the source node the packet is scheduled for transmission with a delay according to the size of the packet. This is necessary to imitate a given bandwidth of the wireless network to ensure that the packet isn't transmitted faster than the network would be able to process. In any case the noise level at each node is incremented by the corresponding signal level of the current packet at the transmission start and decremented at the transmission end. Only if the packet could be received correctly the signal level at the target node is stored instead of raising the noise level. This is needed as another packet could interrupt the transmission which requires the comparison of the signal and noise level.

Figure 4.2 shows a performance test of the provided emulator with the normal ping utility and the flood option which sends as many ICMP Echo Requests (a common network testing protocol) as possible. In this test the emulator calculates a realistic radio propagation model according to chapter 3 and emulates a 1 MBit network. The theoretical minimal round trip time (RTT – the time needed for a complete request and answer) is about 1.6 ms which is always longer in reality because of the processing time of each packet. The Figure shows a value about 1.8 ms for the measured RTT using the emulator which is only about 13% higher than the theoretical minimum. For an even better conformity with a real transmission a small amount of processing time could be added to the timing. These values are not comparable directly to a complete IEEE802.11 net-

work stack because the higher complexity with additional analysis like carrier sensing increases the RTT time.

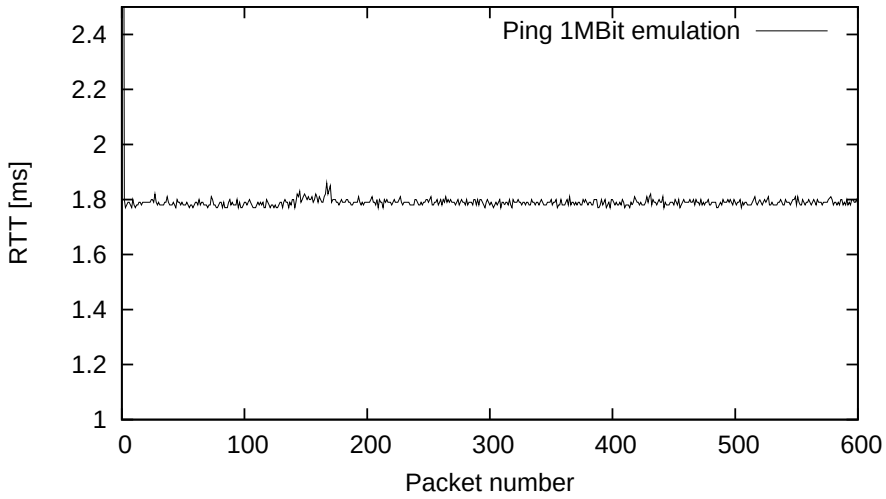


Figure 4.2: Ping performance of the emulator

4.1 Implementation

The emulator is written in plain C++ with the use of different libraries and needs a modified Linux kernel to run properly. As a format for scenario files XML is chosen as it is easy to read and write for human users and can be simply translated into other formats including a better visual display like HTML with CSS using XLST. It holds the information about the scenario environment area, the number of nodes and the positions the nodes should have at a given time. See Appendix B for a complete description of the file format.

4.1.1 Design

One major design goal was the ability to use different emulation models without the need to implement them in the main emulator program. In order to achieve this functionality no emulating routines are implemented in the main program. To start an emulation the

emulator needs to load a emulation engine dynamically which can be defined by a command line parameter. This provides the possibility to create different emulation engines and use them on the same emulator without any further needs despites the development of the specific emulation code. Due to this circumstance it is very easy to add functionality to the emulator and to share these add-ons with other testers or developers. See Appendix A for further details on how to create a custom emulation engine.

The emulator basically consists of one process with three threads running. These threads are the main listener thread, the timer thread and the logging thread. The listener thread receives packets sent through any tap interface, communicates with the kernel, checks the packets and delegates the control to the chosen emulation engine. The timer thread allows the fine-grained scheduling of events to a future time and the logging thread allows to asynchronously write messages to the logfile. Placing the message on a queue at the logthread allows the main emulation thread to continue without a otherwise needed interruption for the time-consuming I/O operations to save the message to disk.

Figure 4.3 shows the usual way of a correctly transmitted packet sent from node X to node Y. The first station on the way is the kernel as it gets the packet directly from the associated tap device. This packet is then delivered to the corresponding userspace application which is in this case the emulator itself. After analysing the packet and extracting source and target node the main process calls the emulation engine module to handle the further treatment of the packet. If the emulation engine decides that the packet could be transfered correctly (i.e. the target node is in range and the noise level is low enough) it schedules the transmission using the timer thread. After the timeout of packet transmission the timer calls again the emulation engine to check whether the transmission was interrupted or did succeed. If the latter is the case the packet finally is handled back to the main process to send the packet through the kernel interface to the right tap device where it is delivered to the listening application.

4.1.2 Main Listener Thread

The main thread opens one TAP device for each virtual mobile node. Every incoming and outgoing traffic comes directly through these TAP devices and is traveling from the userspace application through the kernel to the emulator process or vice versa. Every

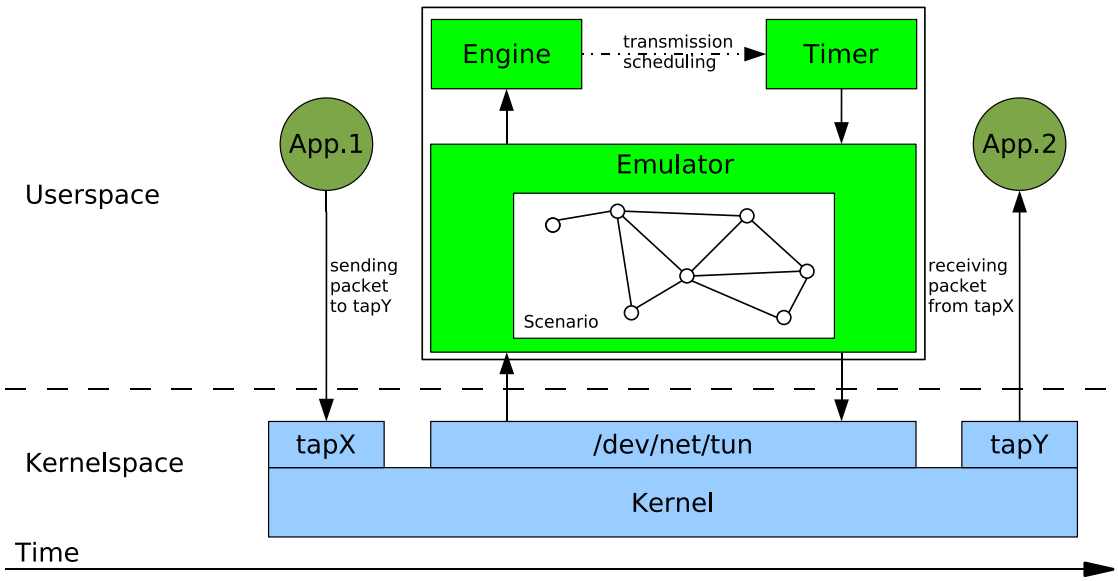


Figure 4.3: The way of a packet from node X to node Y

packet is examined and information about source and target node is extracted to be able to deliver the packet to the correct target TAP device. ARP (Address Resolution Protocol) packets are not handled as normally. Usually an ARP packet is sent to every computer in the target subnet and the correct computer answers the query. The Linux kernel answers to an ARP Packet with the IP of the incoming interface although the target IP address is bound to another interface of the same machine. That behaviour is useful in normal networks but an emulated network on one single machine needs better separation. So in contrast to a real behavior of networks ARP packets are only delivered to the real destination because the Linux kernel answers to an ARP request with the MAC address of the first interface which receives the request regardless if this interface really is bound to the correct destination IP address or one other local interface on the same machine. This might be useful in a real network but because this emulator runs all virtual network interfaces on the same physical machine this behavior ruins the emulation with false MAC–IP pairs.

4.1.3 Timer Thread

One of the major problems was the implementation of a timer thread with a very fine granularity. In order to emulate a wireless network with real latency times the timer

Implementation	min. Timer interval
nanosleep with normal kernel	1 ms
RTC	122 μs
nanosleep with HRTimer patched kernel	70 μs

Table 4.1: Different timer implementations and the corresponding minimal timer interval

must be called at least every 100 microseconds. Different techniques are available to call a timer event in a certain interval. The easiest way is just to call a sleep function like nanosleep. But the vanilla Linux kernel from Linus Torvalds only provides a sleep granularity of about 1 millisecond (at least until the current version 2.6.18). Another approach is the use of the kernel support for the hardware Real Time Clock (RTC). The RTC can be set to the maximum of 8192 Hz. Using this setting every 122 microseconds a RTC interrupt is thrown which is the finest granularity available when using RTC events. But to further improve the grain size a kernel patch called HRTimer [Gle] is used to change the kernel-internal clock mechanism. After applying this patch the kernel allows theoretical timer sleeps down to 1 nanosecond. To achieve a good compromise between timer accuracy and CPU utilisation nanosleep is used with a sleep time of 20 microseconds. Because of other impacts like the process scheduler this leads to a accuracy of about 70 microseconds which might be improved with further kernel scheduler patches. Table 4.1 gives a overview about the different implementations and timings. These measures were taken on an AMD64 box running Gentoo Linux with the kernel version 2.6.18. All testing were done with the here described emulator with a ping test between two TAP interfaces using a minimal emulation engine which scheduled the sending of the packet with a minimal delay of 1 nanosecond. This ensures that the packet gets sent on the very next awaking of the timer thread.

4.1.4 Logthread

In order to interpret the results of an emulation it is necessary to reconstruct the happenings during the emulation. That can only be achieved by analysing a log file which contains all events. As disc input/output (I/O) is very expensive in regard to waiting times it is not advisable to write every log message directly to disc. To avoid this behaviour the logging has been outsourced to a dedicated thread. This implementation allows the other threads to continue with their time-critical work. The logthread writes the data when-

ever the opportunity arises. The best way to implement a asynchronous thread is the producer consumer ringbuffer. In this case multiple producers (the other threads) adding constantly new log messages to a queue while the consumer (the logthread) writes them to disc.

4.1.5 Radio Propagation Model

The example emulation engine provided with this emulator implements a realistic radio propagation model which decides if a packet is received correctly using the noise signal ratio to determine if any collision has happened at the target node. To calculate the noise and signal level at each node for every transmission the distance of each node couple (A,B) is calculated with the information coming from the scenario (Appendix B) file and the current emulation time. (4.2) calculates the position of node A at time τ where P_1 is the last configured position right before time τ, P_2 is the next position configured after time τ, (x_{Pi},y_{Pi}) is the coordinate at position i (i $\in$ {1,2}) and (x_0,y_0) is the calculated position. $\Delta\tau_1$ is the weighted time difference between τ to the time of position P_1 (τ_{P1}) as $\Delta\tau_2$ is the weighted time difference between τ and the time of position P_2 (τ_{P1}).

$$\begin{pmatrix} \Delta\lambda_1 \\ \\ \Delta\lambda_2 \end{pmatrix} = \begin{pmatrix} \frac{\tau-\tau_{P1}}{\tau_{P2}-\tau_{P1}} \\ \\ \frac{\tau_{P2}-\tau}{\tau_{P2}-\tau_{P1}} \end{pmatrix} \tag{4.1}$$

$$\begin{pmatrix} x_0 \\ \\ y_0 \end{pmatrix} = \begin{pmatrix} x_{P1} \cdot \Delta\tau_2 + x_{P2} \cdot \Delta\tau_1 \\ \\ y_{P1} \cdot \Delta\tau_2 + y_{P2} \cdot \Delta\tau_1 \end{pmatrix} \tag{4.2}$$

Having calculated both positions of node A (x_1,y_1) and B (x_2,y_2) the distance d is calculated according to common vector distance calculation (4.3).

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{4.3}$$

With this distance the signal power is calculated as described in Chapter 3 using the Free-Space Loss and the Two-Ray Ground model. Because dB is a logarithmic unit it is not possible to simply add multiple signal levels to get the complete noise level. Instead it is necessary to convert each value back to a linear unit, add the new noise and convert back to a logarithmic value.

$$noise_{total} = \log_{10}(10^{noise_{total}/10} + 10^{noise_{new}/10}) \tag{4.4}$$

When a packet is sent from node A to node B the signal level is calculated using the formulas above and is compared to the current noise level at the target node. If the difference is higher than a given limit the packet is considered to be able to be received. But if the noise level increases during packet delivery the signal noise ratio is checked again and may brake the transmission. Only if the transmission ends without any interrupts the packet is delivered to the corresponding node. Then the noise levels of all nodes are decremented again.

4.2 Kernel requirements

4.2.1 Send-to-Self Patch

Every emulated wireless node has an opened TAP device which needs the TUN/TAP-Driver enabled in the kernel configuration. But because of optimizations of the current Linux kernel every IP packet which destination is an IP address bound to a local network interface gets routed directly to the specific interface avoiding OSI level 2 (see Figure 4.4). These packets will never reach the emulator. In order to receive packets from internal routing at the opened TAP devices the kernel needs to be constrained to send each packet through the complete IP chain including physical layer which is in our case the emulator. This is done by the Send-to-Self (STS) patch originally developed by Ben Greear [Gre]. After applying this patch the kernel treads every packet equally.

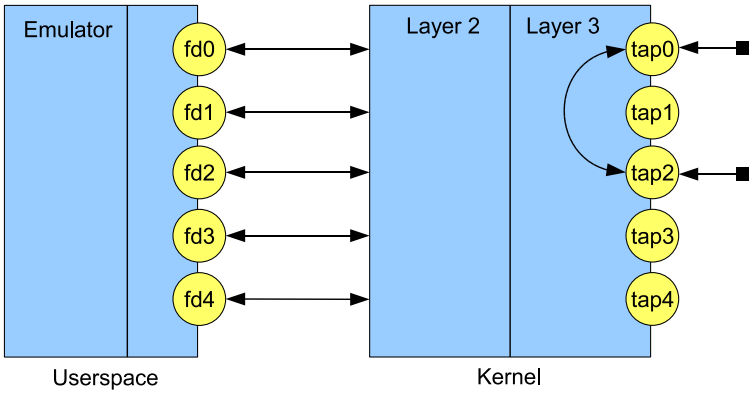


Figure 4.4: Kernel packet routing without STS patch

4.2.2 HRTimers Patch

One major benefit of using an emulator instead of a simulator is the use of realtime information. But to accomplish realtime measurement a timer with fine granularity is needed. The normal Linux kernel only provides a granularity of about 1 millisecond. The hrtimers patch developed by Thomas Gleixner [Gle] enables a timer interval of about 70 microseconds usable with normal glibc functions like "nanosleep".

4.3 System requirements

The emulator uses external libraries for a better and faster development. Therefor these libraries have to be installed on the system to run the emulator. For the thread implementation the Gnu Common C++ [CPP] library is used. GNU Common C++ is a framework with class support for easy developing of thread enabled applications. Another important library is Libxml++ [JdC]. This library is a C++ wrapper for the libxml XML parser library. This library is used to parse the scenario XML files using the DOM method. This provides a fast integration of XML-based files for storing information about the scenario.

4.4 Requirements for testing applications

Because of the special nature of this emulator using TAP devices for representation of each node every application must open each socket using the SO_BINDTODEVICE option to ensure that the application only listens and send through the chosen TAP device. This is very critical due to the fact that all processes for all nodes are running on the same physical machine and without this option each port could only be opened by one process at a time and because all interfaces are local interfaces the process would get every packet directed for this port. The only avoidance of this restriction is a completely separated environment like the paravirtualization used by MarNET or a virtual environment used by a simulator. As the intention of this thesis was the development of an emulator a simulator is no alternative and with paravirtualization the latency would be to high for a complex radio propagation model. The use of TAP devices on only one instance of an operating system offers the best possibilities for fine-grained timers coupled with least overhead. In contrast the modifications to testing applications should be rather simple and many applications already included this functionality. So the benefits of this limitation should outweigh the drawbacks.

Chapter 5

Performance Analysis

In order to evaluate the performance of the new emulator it is compared to the MarNET emulator. The latency as well as the network throughput are compared. The results provide good hints about the abilities the emulator could offer. The testing was done on a single core AMD64 computer. In each case the emulated network consists of 4 nodes. For the tests the emulation functionalities were reduced to the minimum. In the MarNET emulator only the virtual operating systems (XEN DomUs) were started without the real emulation unit. In this mode each node is able to send to every other node without any delay. On the other hand LoLaWe used a minimal emulation engine which scheduled the packet delivery to the very next timer awaking without any further simulation or testing. Using these simplifications the results should provide the best performance of each emulator.

Latency testing was done by the simple network analysing tool "ping" which sends ICMP Echo Requests for response time measurement. In both emulators the testing assumed a sending interval about 0.1 ms. During the measurement another ping was sent between two independent nodes to simulate the impact of a minimal network load. Figure 5.1 indicates that both emulators are capable of short RTT times. LoLaWe is quiet stable around 0.05 ms with few short peaks while MarNET in contrast suffers from many and rather high peaks. The statistics of table 5.1 clearly prove that LoLaWe is the better performing system regarding dependable constant timings for a more realistic emulation.

The network testing tool "NetIO" [NET] was used for the measurement of the network throughput. NetIO opens a server at one end of the communication channel and the same

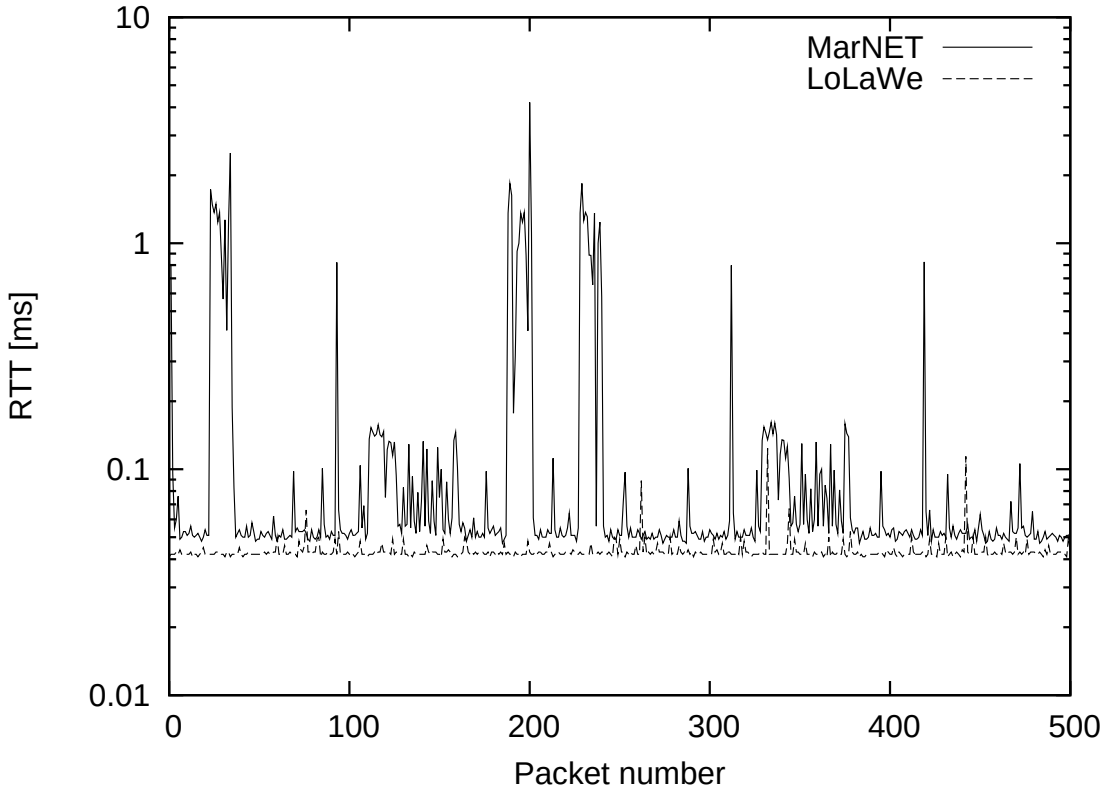


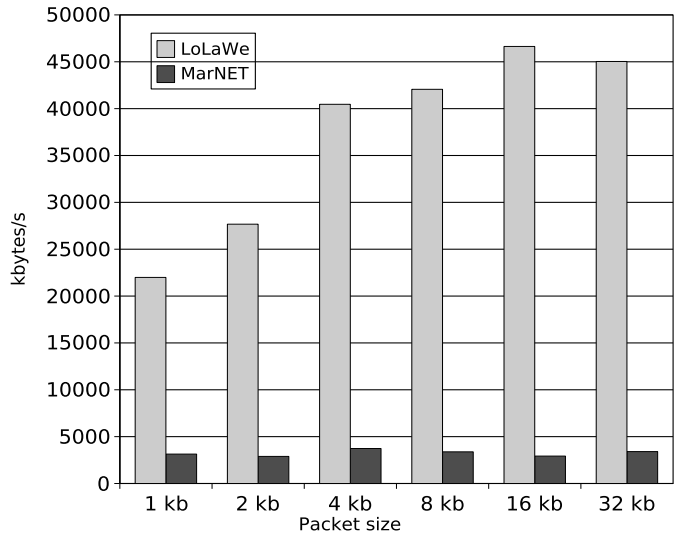
Figure 5.1: Ping comparison of MarNET and LoLaWe

Emulator	Mean value	standard deviation
MarNET	0.1585	0.3680
LoLaWe	0.0434	0.0058

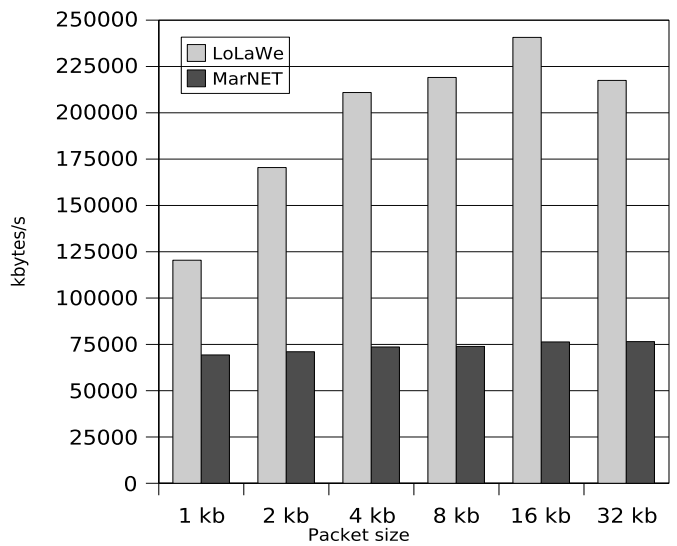
Table 5.1: Mean value and standard deviation of the RTT measurement (Figure 5.1)

program is used as the client. It is possible to test UDP as well as TCP packets and by default different packet sizes are used. Figure 5.2 shows the results of the UDP and TCP throughput measurement. The TCP throughput of LoLaWe is about three times faster than MarNET. UDP packets are even sent ten times faster. Notable are the differences between various packet sizes. While LoLaWe performs better with increasing packet sizes (except the 32 kbyte packets) MarNET stays more or less at the same level. The big performance differences between UDP and TCP communications in both emulators need further investigation but in both cases it is very clear that LoLaWe easily outperforms MarNET by orders of magnitude. While MarNET seems to be unable to emulate exhaustive UDP traffic in a 100 MBit network LoLaWe should be able to emulate even a gigabit network. As both emulators are designed to emulate mobile ad-hoc networks

with considerably lower bandwidths these results are of less significance.



(a) UDP Throughput



(b) TCP Throughput

Figure 5.2: UDP and TCP throughput comparison of MarNET and LoLaWe

Chapter 6

Conclusion

The presented emulator is a hybrid of a complete simulation environment and a native testing environment with real hardware. It supports very low latency timer intervals below 100 μs which enables the simulation of real-time wireless behavior like transmission delays and collision detection. The complete emulation runs on a single computer to be as cost-effective as possible. The possibility of loading different emulation engines makes it very easy to implement different kinds of techniques. For example a complete IEEE802.11 MAC layer could be compared to a simple ALOA layer or any other wireless communication protocol. In order to emulate non-existing hardware features it is possible to integrate these features in a modified TUN/TAP driver and support them in the emulator with only little changes. This emulates needed features with a real ready-to-use kernel implementation which can be used later on for the complete implementation of the specific driver or protocol. The performance of the emulator with its fine-grain timer intervals enables the emulation of complex wireless protocols with short backoff times and fast transmission rates as seen in the performance tests.

The combination of a real implementation, a real operating system and a simulated realistic radio propagation model could represent wireless communications more realistic than other emulators or simulators. This emulator could be used as a second testing step when there is no possibility to use real hardware as it is the case with the CXCC protocol. The results can be compared with the simulator results to measure the impact of the real operating system and tools. But of course simulation or emulation results can't replace reality testing.

Bibliography

[AO96] M. Allman and S. Ostermann. ONE - the Ohio Network Emulator. 1996.

[BSR+05] O. Battenfeld, M. Smith, P. Reinhardt, T. Friese, and B. Freisleben. A modular routing architecture for hot swappable mobile ad hoc routing algorithms. In *Proc. of the Second International Conference on Embedded Software and Systems, Xian, China*, pages 359–366. Springer-Verlag, 2005.

[CPP] GNU Common C++ - http://www.gnu.org/software/commoncpp/.

[CS03] Mark Carson and Darrin Santay. Nist net: a linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003.

[Fri46] H. T. Friis. A note on a simple transmission formula. *Proc. IRE*, 34, 1946.

[FTO01] Juan Flynn, Hitesh Tewari, and Donal O'Mahony. JEmu: A Real Time Emulation System for Mobile Ad Hoc Networks. In *Proc. First Joint IEI/IEE Symp. Telecomm. Systems Research*, Dublin, Ireland, November 2001.

[Gle] Thomas Gleixner. High Resolution Timer - http://www.tglx.de/hrtimers.html.

[Gre] Ben Greear. Sent-to-Self patch - http://www.ssi.bg/~ja/#loop.

[JdC] Ari Johnson, Christophe de Vienne, and Murray Cumming. libxml++ - http://libxmlplusplus.sourceforge.net/.

[JS] Glenn Judd and Peter Steenkiste. Using Emulation to Understand and Improve Wireless Networks and Applications -

http://www.cs.cmu.edu/~glennj/JuddNSDIEmulator.pdf. In *NSDI 2005*.

[LNH+04] Qiong Luo, Lionel M. Ni, Bingsheng He, Hejun Wu, and Wenwei Xue. MEADOWS: Modeling, Emulation, and Analysis of Data of Wireless Sensor Networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 58–67, New York, NY, USA, 2004. ACM Press.

[MHR05] Steffen Maier, Daniel Herrscher, and Kurt Rothermel. On Node Virtualization for Scalable Network Emulation. In *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '05), Philadelphia, PA, July 24–28, 2005*, pages 917–928. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Simulation Councils, Inc., Juli 2005.

[MRBV04] P. Mahadevan, A. Rodriguez, D. Becker, and A. Vahdat. MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks - http://ramp.ucsd.edu/~pmahadevan/publications/mobinet-mc2r.pdf. In *UCSD Technical Report CS2004-0792*, July 2004.

[NET] NetIO - http://www.ars.de/ars/ars.nsf/docs/netio.

[NS2] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns.

[PAR] Paravirtualization - http://en.wikipedia.org/wiki/Paravirtualization.

[PP05] Metija Puzar and Thomas Plagemann. NEMAN: A Network Emulator for Mobile Ad-Hoc Networks. In *8th International Conference on Telecommunications (ConTEL 2005)*, Zagreb, Croatia, June 2005.

[RAD] Radio Propagation Models in the NS-2 - http://www.isi.edu/~weiye/pub/propagation_ns.ps.gz.

[SHF05] M. Smith, S. Hanemann, and B. Freisleben. Coupled simulation/emulation for cross-layer enabled mobile wireless computing. In *Proceedings of the Second International Conference on Embedded Software and Systems, Xian, China*, pages 375–383. Springer-Verlag, 2005.

[SLM07] Björn Scheuermann, Christian Lochert, and Martin Mauve. Implicit Hop-by-Hop Congestion Control in Wireless Multihop Networks. In *Elsevier Ad Hoc Networks*, 2007.

[WLZ+04] Hejun Wu, Qiong Luo, Pei Zheng, Bingsheng He, and Lionel M. Ni. Accurate Emulation of Wireless Sensor Networks. In *NPC*, pages 576–583, 2004.

[XEN] XEN - http://www.cl.cam.ac.uk/research/srg/netos/xen/.

[ZL02] Yongguang Zhang and Wei Li. An Integrated Environment for Testing Mobile Ad-Hoc Networks. In *MobiHoc '02: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, pages 104–111, New York, NY, USA, 2002. ACM Press.

Appendix A

Create a Module

The emulator allows the dynamic loading of emulation engine modules. The creation of a new module is easy but has to be compliant with some premises. The first and important requirement is that the main module class needs to be derived from the abstract class "VirtualWLAN" (see file virtualwlan.h). The new class needs to reimplement the method "sendPacket" in order to change the way of packet handling which is called every time a packet should be send from one node to another. For a full control of the packet handling it is necessary to implement extended IPPacket and Node classes. All classes should be created in the "lolawe" namespace to keep things together.

Listing A.1: Minimal module

```
1  #ifndef VWLAN_EXAMPLE_H_
2  #define VWLAN_EXAMPLE_H_
3  #include <virtualwlan.h>
4
5  namespace lolawe {
6
7  class VWLAN_example: public VirtualWLAN {
8  public:
9          VWLAN_example();
10         virtual ~VWLAN_example();
11         bool sendPacket(IPPacket *packet) { p->send(); } /* always
               send the packet immediately */
12 };
13
14 }
15 #endif /*VWLAN_EXAMPLE_H_*/
```

The second necessary thing is the loader method. To be able to load a shared object during runtime the dlopen C-funcion is used. Because exported C++ method symbols are named according to the class it is impossible to directly call C++ methods with dlopen. In order to get a correct instance of the created class a wrapper function has to be provided as follows:

Listing A.2: C wrapper functions

```
1 extern "C" {
2         VirtualWLAN* create() {
3                 return new VWLAN_example();
4         }
5
6         void destroy(VWLAN_example* v) {
7                 delete v;
8         }
9 }
```

The emulation engine shall use the provided objects for communication with the main thread, the log thread and the timer thread. In detail the following classes are provided:

Listing A.3: The Virtual WLAN class

```
1 class VirtualWLAN: public ObjectFactory {
2 public:
3         VirtualWLAN() {};
4         virtual ~VirtualWLAN() {};
5         virtual bool sendPacket(IPPacket *packet) = 0;
6         virtual bool receivePacket(IPPacket &p) = 0;
7         virtual bool initialise(char *configfile) = 0;
8         virtual void setScenario(Scenario *s) { scenario = s; };
9         virtual void setEventTimer(EventTimer *t) { timer = t; };
10        virtual void setLogThread(LogThread *l) { logthread = l; }
11        virtual long getCurrentSecs() { return timer->getCurrentSec();
              }
12        virtual void start() { }
13 protected:
14        Scenario *scenario;
15        EventTimer *timer;
16        LogThread *logthread;
```

Listing A.4: The Scenario class

```
1 class Scenario {
```

```
public:
        Scenario(ObjectFactory* f,string sfile);
        virtual ~Scenario();
        Node* getNode(int n) { return nodes.at(n); }
        int getNodesCount() { return nodes.size(); }
        int getHeight() { return height; }
        int getWidth() { return width; };
        int getBandwidth() { return bandwidth; };
        string getName() { return name; };
        string getDescription() { return description; };
```

Listing A.5: The Node class

```
class Node: public Eventable {
public:
        Node(int no);
        virtual ~Node();
        void addPosition(int time, int x, int y);
        void setDescription(string desc) { description = desc; };
        string getDescription() { return description; }
        int getNumber() { return number; }
        Position* getPositionObject(int n) { return positions.at(n); }
        Position* getPosition(int time);
        int getDistance(Node *node, int time);
        void callMethod(int i, long time);
protected:
        vector<Position*> positions;
        int number;
        string description;
        int lastTimePos;
        Position* getLowerPos(int time);
        Position* getHigherPos(int time);
```

Listing A.6: The IPPacket class

```
class IPPacket: public Eventable {
public:
        IPPacket(TapSelector* tap,char *raw, int len);
        virtual ~IPPacket();
        int getSourceTapNo() { return sourceTapNo; }
        int getTargetTapNo() { return targetTapNo; }
        int getPacketType() { return packetType; }
        int getDataLen() { return data_len; }
        char *getRawData() { return rawData; }
```

```
10          TapDevice* getTargetTapDev() { return tapsel->getTapDevice(
                targetTapNo); }
11          TapDevice* getSourceTapDev() { return tapsel->getTapDevice(
                sourceTapNo); }
12          void send();
13          void callMethod(int i, long time);
```

Listing A.7: The Timer class

```
1 class EventTimer : public virtual Thread {
2 public:
3           EventTimer();
4           virtual ~EventTimer();
5           void run();
6           void stop();
7           void addItem(Eventable *it, int par, long time);
8           long long getTime();
9           long getCurrentSec();
```

Listing A.8: The Logging class

```
1 class LogThread: public ThreadQueue {
2 public:
3           LogThread();
4           virtual ~LogThread();
5           void addMessage(int level, string msg);
6           void runQueue(void *data);
7           void startQueue(void);
8           void stopQueue(void);
9           void setLogLevel(int i) { logLevel = i; };
```

Listing A.9: The Object Factory class

```
1 class ObjectFactory {
2 public:
3           virtual ~ObjectFactory() {};
4           virtual void setTapSelector(TapSelector *t) { tapsel = t; };
5           virtual Node* createNode(int no) { return new Node(no); }
6           virtual IPPacket* createIPPacket(char *raw, int len) { return
                new IPPacket(tapsel ,raw, len); }
7 protected:
8           TapSelector *tapsel;
```

Appendix B

Scenario File Format

The scenario file is a XML file which sets the size of the emulation area, the number of nodes and all positions of each node. All time values are seconds and all position values are meters. The DTD for the XML file:

Listing B.1: DTD for Scenario files

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated from Scenario_Example.xml by XMLBuddy -->
<!ELEMENT scenario (description?,node+)>

<!ENTITY % nonNegativeInteger "NMTOKEN">
<!-- <!ENTITY % nonNegativeInteger " datatype CDATA #FIXED '
    nonNegativeInteger'"> -->

<!ATTLIST scenario
        height %nonNegativeInteger; #REQUIRED
        time %nonNegativeInteger; #REQUIRED
        width %nonNegativeInteger; #REQUIRED
        bandwidth %nonNegativeInteger; #REQUIRED
        name CDATA #REQUIRED
>
<!ELEMENT description (#PCDATA)>
<!ELEMENT node (description?,position+)>
<!ATTLIST node
        no %nonNegativeInteger; #REQUIRED
        x %nonNegativeInteger; #REQUIRED
        y %nonNegativeInteger; #REQUIRED
>
```

```
<!ELEMENT position EMPTY>
<!ATTLIST position
        time %nonNegativeInteger; #REQUIRED
        x %nonNegativeInteger; #REQUIRED
        y %nonNegativeInteger; #REQUIRED
>
```

Listing B.2: Example Scenario files

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE scenario SYSTEM "LoLaWe_Scenario.dtd">
<scenario width="1000" height="1000" time="2000" name="Test Scenario 1
    " bandwidth="1000">
        <description>This is the description for the first example for
            a scenario file.</description>
        <node no="1" x="10" y="10">
                <description>This is the description for a node.</
                    description>
                <position time="50" x="40" y="100" />
                <position time="150" x="50" y="130" />
        </node>
        <node no="2" x="24" y="234">
                <position time="50" x="140" y="100" />
                <position time="150" x="650" y="30" />
        </node>
        <node no="3" x="10" y="10">
                <position time="50" x="50" y="30" />
                <position time="150" x="80" y="40" />
        </node>
</scenario>
```

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 5. Januar 2007 Matthias Jansen