RWTH AACHEN
RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

Distributed Systems

A Hybrid Emulation Environment for Wireless Networks

Rheinisch-Westfälische Technische Hochschule Aachen
LuFG Informatik 4 Verteilte Systeme

Diploma Thesis

Hendrik vom Lehn

Advisors:

Dipl.-Inf.	Elias Weingärtner
Prof. Dr.-Ing.	Klaus Wehrle

Registration date:	05.	January	2010
Submission Date:	05.	July	2010

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Aachen, den 5. Juli 2010

Kurzfassung

Aufgrund der Eigenschaften eines drahtlosen Übertragungskanals ist die Entwicklung von Software für drahtlose Netzwerke eine schwierige Aufgabe. Echte Experimente sind mühselig in der Ausführung, Netzwerksimulationen hingegen abstrahieren zu sehr von echten Systemen. Netzwerkemulation stellt einen Mittelweg zwischen diesen beiden Techniken dar und versucht deren spezifische Vorteile zu kombinieren. Derzeit verfügbare Lösungen für die Emulation von drahtlosen Netzwerken sind jedoch zu unflexibel oder stellen der zu testenden Software keine authentische Umgebung zur Verfügung. Diese Arbeit nimmt sich diesem Problem durch den Entwurf und die Entwicklung einer virtuellen drahtlosen Netzwerkkarte an. Mit dieser virtuellen Netzwerkkarte ist es möglich, unmodifizierte Software in simulierten drahtlosen Netzwerken auf eine flexible Art zu untersuchen.

Abstract

Due to characteristics of the wireless channel, software development for wireless networks is a difficult task. While real experiments are cumbersome to perform, network simulations abstract too much from real implementations. Network emulation, as a middle way between these two methodologies, aims at a combination of their specific advantages. Current implementations for wireless network emulation, however, lack flexibility or do not provide an authentic environment to the software under test. This thesis addresses these issues through the design and implementation of a virtual wireless network card which allows the flexible evaluation of unmodified software in simulated wireless networks.

Acknowledgments

Looking back at the process of writing this thesis, I received valuable support from a number of people in many different ways. For this I would like to thank:

Prof. Dr. Wehrle and Prof. Dr. Gross, for being my supervisors of this diploma thesis and for taking their time for discussions.

My advisor, Elias Weingärtner, for suggesting this interesting topic, his useful advice and for always having time for my questions.

All students and employees of the Distributed Systems Group, for the nice work atmosphere and valuable conversations. In particular, I would like to thank René Hummen and Florian Schmidt for the many helpful discussions.

Dartmouth College, for the provision of wireless experimentation data as part of the CRAWDAD archive.

Andreas Kehlenbach, Michael Kozielski, Martin Henze and my sister Antje, for their help in proof-reading this document.

My girlfriend, Kirstin, for her moral support and understanding.

Last but not least, my parents, for always supporting me during the time of my studies.

Contents

1 Introduction **1**

2 Background **3**

2.1 Communication Systems Engineering 3

2.2 Network Emulation 5

2.3 Wireless Networks 7

2.3.1 Challenges of WLANs 7

2.3.1.1 Physical Layer 7

2.3.1.2 Medium Access Control 9

2.3.1.3 Compatibility to Ethernet 11

2.3.2 IEEE 802.11 11

2.3.2.1 Network Types 12

2.3.2.2 Physical Layer 13

2.3.2.3 Medium Access Control 14

2.3.2.4 Integration into Ethernet Networks 15

2.3.3 Testing in Wireless Networks 17

2.4 Real Implementations 17

2.4.1 Operating System 17

2.4.1.1 Network Driver 18

2.4.1.2 Driver Interface 18

2.4.1.3 Influence on Testing 19

2.4.2 Testbeds 19

2.4.3 Virtualization 20

2.4.3.1 The Xen Hypervisor 21

2.4.3.2 Testing of Communication Systems 21

2.5 Network Simulation 22

2.5.1 Evaluation of Communication Systems 23

2.5.2 Simulation as Network Emulator 23

3 Conceptual Design **27**

3.1 Objective 27

3.2 Approach 29

3.2.1 The Overall System 29

3.2.2 The Split Network Stack 30

3.2.3 Design Alternatives 31

3.3 Emulated Nodes 32

3.3.1 Network Simulation 33

3.3.2 Real System 33

3.3.3 The Connection 35

4 Implementation **39**

4.1 Functionality 40

4.2 Message Exchange 42

4.2.1 Transport Mechanisms 44

4.2.2 UDP Transport 44

4.3 Extending ns-3 45

4.4 Linux Driver 48

4.5 Setup Example 51

5 Evaluation **53**

5.1 Basic Measurements 54

5.1.1 Comparison to Real Measurements 54

5.1.2 Influence of Virtual Machines 58

5.1.3 Detailed Time Analysis 61

5.2 Outdoor Scenario 63

5.3 Summary 68

6 Related Work **71**

6.1 Hardware Emulation 71

6.2 Software Emulation 73

6.3 Emulation through Simulation 76

7 Future Work **79**

7.1 Further Evaluation 79

7.2 Additional Functionality 80

7.3 Application Areas 82

8 Conclusion **85**

Bibliography **87**

A Appendix **97**

A.1 Usage Howto 97

A.2 Additional Figures 100

1 Introduction

An important step during the development of a communication system is to evaluate the system and to verify that all its components provide the intended functionality. In order to do so, prototypes can be used for the evaluation of systems in real experiments. Due to the distributed nature of communication systems, such a prototype has to consist of several instances. This makes the evaluation of larger communication systems a difficult and costly task. In case of communication systems which involve wireless networks, the evaluation is even more complicated, because the physical placement of nodes has an influence on the communication behavior. This requires to spread the communication system over large areas and, in case of mobility, to physically move components.

In research, the main interest is often not to develop the actual communication system itself, but rather to learn more about certain aspects of its behavior. A therefore often used evaluation methodology is network simulation. Through modeling all components of a communication system as part of a computer program, it is possible to examine how the system behaves in arbitrary communication scenarios. While this allows for an easier evaluation than the use of prototypes, it can also lead to inaccurate results if the models used abstract too much from real implementations. A further drawback of simulations is that it is in general not possible to use the same program code to build a simulation model and a real implementation which might be used in later development phases.

The focus of this thesis is the evaluation process of software which makes use of wireless networks. In this case, the use of prototypes means to utilize a number of computers, which are equipped with wireless network devices, to execute the software under test. Consequently, the hardware-requirement itself and deploying the devices in the right way to form a desired network topology are the most troublesome parts of this process. If one, on the other hand, uses network simulations, the wireless network itself exists as a ready simulation model, but it is not easily possible to use the developed software as part of the simulation.

In order to bridge the gap between these different evaluation methodologies, this thesis proposes a combination of both: A network simulation is used to model a wireless network to which real software implementations are connected. This technique, which is a form of wireless network emulation, thus allows to execute the software under test without having the trouble of real wireless networks. While many network simulation tools provide ready support for the data exchange between the simulation and real systems, existing solutions do not provide an authentic wireless environment, so that the software under test has to be modified or cannot make use of the wireless network in its full functionality.

The concept which has been developed within this thesis aims to provide an authentic wireless network environment to the software under test. To that end, a wireless network card driver is used as part of an otherwise unmodified operating system which executes the software under test. This driver, however, does not connect to a real wireless network card, but to a wireless network card that is modeled as part of a network simulation. Thereby, the software under test can access the virtual wireless network card using the genuine interfaces which are provided by the operating system. The network simulation is used mainly to model the interconnecting wireless network, but can also include entirely simulated stations that are not realized through real systems, hereby leading to hybrid emulation.

As a concrete implementation of this concept two major components have been developed: An 802.11 [45] wireless network card driver for Linux [102] and a wireless emulation bridge for the ns-3 [103] network simulator. The wireless emulation bridge is installed on ns-3's model for 802.11 network cards and exports its functionality to the driver which is used as part of the real system. The network simulation and driver communicate using a message exchange format which has been developed based on UDP [82]. Through the use of an existing network connection, it is possible to run the driver (and therefore also the software under test) together with the network simulation, in virtual machines or distributed across a network of real computers.

By combining the newly developed components with already existing simulation models, regular operating systems and virtual machines, a very flexible environment for the evaluation of wireless network software is created.

The remainder of this thesis is structured as follows: Chapter 2 provides background information on several topics that are important for this thesis. The developed concept for wireless network emulation and its implementation are described in Chapters 3 and 4. Using this implementation, a number of experiments have been performed. A description of these experiments and the corresponding results can be found in Chapter 5. Subsequently, in Chapter 6, related approaches that also aim at the evaluation of wireless networks are introduced and compared to the approach of this thesis. Chapter 7 shows some areas in which the developed emulation framework could be further improved or extended. Finally, Chapter 8 gives a short summary and draws conclusions from what has been shown throughout this thesis.

2

Background

The concept proposed by this thesis uses different techniques which require some background knowledge in communication systems engineering, wireless networks and network simulation. The goal of this chapter is to give an introduction into these fields. This document however, cannot start from scratch, so that some background knowledge in computer networks is necessary, provided for example in literature such as [99].

Section 2.1 introduces the context of this thesis and explains why testing is important for the process of communication systems engineering. Network emulation in its generic fashion is introduced in Section 2.2. It is followed by an introduction to wireless networks in Section 2.3. Afterwards real implementations (Section 2.4) and network simulation (Section 2.5) are introduced.

2.1 Communication Systems Engineering

The common ground of all communication systems is that peers communicate with each other over a physical medium. Usually, such systems are implemented as several components which together make up the communication peer. This has the advantage that the components can be developed independently from each other and can also be combined in different ways to create different communication systems. In the case of computer networks, one usually views these components as a layered system, often called *network stack*. The *International Standards Organization* [53] defined the *Open Systems Interconnection* reference model [51] (see Figure 2.1), often referred to as ISO OSI model. Although it does not match actual implementations of e.g. the Internet perfectly, it is almost always used to describe the functionality of communication system components. The idea of this layered model is that two applications communicate to each other by exchanging messages which traverse all layers of both sender and receiver. Each layer modifies the message in a certain way and hands it over to the next layer by using a well-defined interface. The advantage

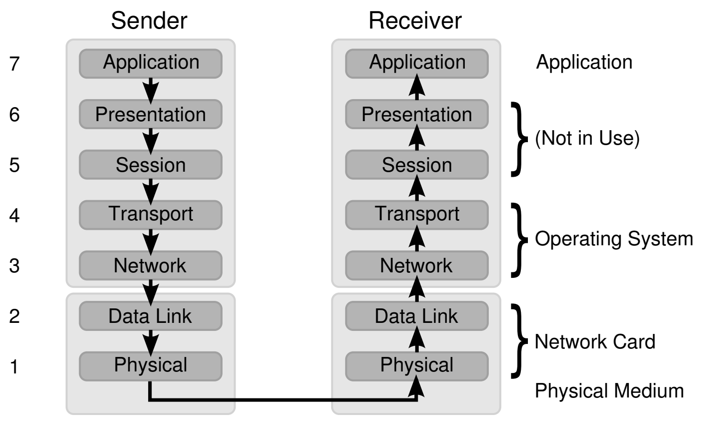


Figure 2.1 A message is sent through the layers of the ISO OSI reference model.

of such an approach is that the implementation of a layer can be exchanged without modifying the other layers. A more detailed description of the functionality of the single layers can be found in [99].

Most of the layers in a modern communication system are pure software. The network and transport layers are usually part of the operating system. Software of the application layer is usually run as an application, but can in special cases also be part of the operating system. Only the lower two layers are usually implemented as separate hardware (the *network card*). But even for those two layers, software is gaining more and more importance: More complex data link layers are implemented in software (as *firmware* running on the network card) or this task is transferred to the network card's driver (so-called *SoftMAC* [31]). And with *software defined radio* [75] software is even used for the physical layer.

From a software engineering perspective one can view the development of a layered communication system as a *component-based software process* [96], where all layers except the one to be developed are existing components which are re-used. Such *software process models* are abstract views on the different steps in the process of software engineering. They include tasks like specification, design, implementation and testing. The perhaps most famous one is the *waterfall model* [90, 96] in which each of the steps follows sequentially one after the other. In contrast, *evolutionary development* [96] allows jumping back and forth between these steps, thus making quick changes possible without having to repeat the whole process. What all models have in common is that testing occurs throughout different phases of the development process. Single components are already being tested during the implementation phase and the complete system has to be tested after all components have been integrated. The use of testing is to verify that the software has the specified behavior and to validate that this behavior is indeed the desired one.

Such software process models are mostly tailored towards the development of products. In case of research, things might be a bit different: The primary goal of the researcher is in most cases not the system itself. Instead, he uses it to gain additional knowledge or simply to prove that it is possible to build the system in a specific way (*proof of concept*). The developed system is then a *research prototype* which offers the needed functionality, but does not need to have such high quality as a real product would have. Similarly, a first step in software processes can be the use of *throwaway prototypes* [96] to get a better understanding of a problem and its solution.

Regardless of whether the developer sticks to specific software process models or not, it is always necessary to run the system in order to test and evaluate it. In case of a communication system there is a difficulty involved: It is not just one instance of the software which is running, but multiple instances distributed across multiple hosts. While this poses no problem for a regular use of the software (it is even its feature), it is an additional barrier for testing and evaluation. The software has to be run and monitored synchronously on multiple hosts, which is more difficult both in an automated and manual fashion. It requires more resources by means of hardware, personnel and time.

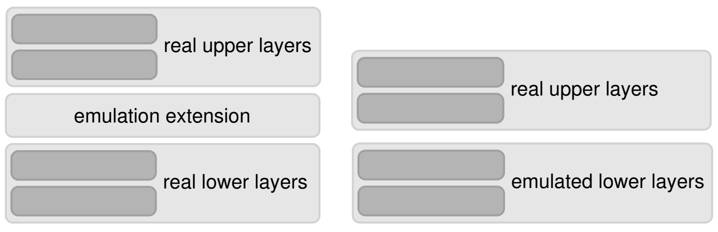
In cases where only the behavior of a hypothetical communication system is of interest, *network simulation* can be a way out of these problems. In network simulations, one does not build the system itself, but uses a software which mimics the behavior of the system's components. In short, the advantages of this method are its ease of use, the high level of control and the possibility to repeat experiments under the exact same conditions. Drawbacks are a potential lack of realism and in many cases the requirement of separate simulation code. The topic of network simulation will be covered in more detail in Section 2.5.

No matter which method is used, it is always important for the testing of a communication system to regard it as a whole: Both the multiple instances required for communication and the composition of multiple building blocks in each of those instances. Due to the layered structure, a developer usually only implements a small part of the communication system, but for testing the system, it is required to use the complete system. So what makes testing of communication systems difficult is mostly not to run the developed piece of software itself, but to create the environment which the developed code is supposed to run in.

The above applies to *network protocol* implementations as well as *applications*. In the context of communication systems, these terms can have different meanings: Application can refer to an implementation with application layer functionality, or it can mean that the software is running in the application context of an operating system. In many cases, both meanings coincide, but this is not necessarily the case. A network protocol can in principle be located in any layer, including the application layer. However, this does not mean that every application accessing the network functionality of the operating system is a network protocol implementation. Irrespective of which concrete meaning the terms network protocol implementation and application have in a specific case, they always have to be tested in the same manner as part of the complete communication system. Both terms will therefore be used in the remainder of this document.

2.2 Network Emulation

Network simulation and the use of real implementations are two different methodologies which both have their specific advantages. *Network emulation* brings both



(a) Network emulation using the complete network stack. (b) Network emulation replacing some layers.

Figure 2.2 Two different methods of network emulation.

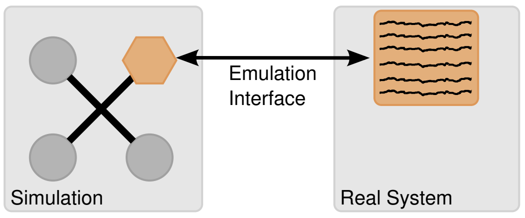
worlds together by using the implementation under test in a partially synthetic context. A more precise definition of network emulation is hard to give, since there is a diverse variety of techniques which are covered by this term.

Early approaches of network emulation make use of almost unmodified communication systems and just alter the delivery of packets by hooking in at some point in the network stack (see Figure 2.2(a)): There are tools which act as a proxy of the Berkeley socket API [50], hook in between the network and transport layer [89] or utilize the interface between the network layer and the network card's driver [3, 76]. What they have in common, is that they do not interpret the content of the packets they intercept (so-called *opaque* network emulation [26]). All they do is delay the delivery of packets, drop them, reorder the delivery or add errors. Through modeling these effects, it is then possible to test the behavior of implementations in more complex networks, such as wide-area networks [50, 3] or mobile networks [76], by using just a regular local area network. While some tools only allow a configuration through static parameters [3], others use more sophisticated methods, such as the use of recorded traces, in order to emulate a network with the same properties [76].

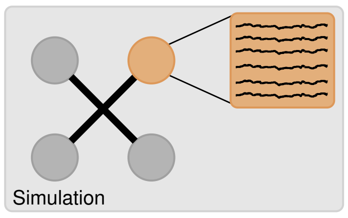
What the abovementioned tools have in common is that they make use of the complete network stack by acting as a proxy at some layer-boundary. Apart from that methodology, there also exist emulation systems which replace the functionality of whole layers of a communication system (see Figure 2.2(b)). One example of the latter is *Neman* [84], a network emulation tool which provides virtual network devices to the operating system. It switches all packets sent over these interfaces by itself and thereby replaces both the physical and the data link layer. An advantage of such an approach is that the included layers can be modeled in any abstract way and thus no full-featured implementation is needed.

A popular way to implement the replaced layers is the use of a network simulation tool. Simply put, a network simulation is run and communicates with a real implementation through a special node inside the simulation which acts as gateway. Since the internal techniques of simulations differ from those of regular software, some conversions regarding time concepts and exchanged data have to be performed. One of the first implementations of this concept, the problems arising and their solutions are described by Fall in [26]. A more detailed description of network emulation using simulation will be given in Section 2.5.

A slightly different variant of network emulation is *environment emulation* ([26]). While the former approaches run the real implementation in its genuine environment and connect it to an emulated network, environment emulation builds an artificial



(a) Network Emulation



(b) Environment Emulation

Figure 2.3 Two concepts connecting a network simulation with real implementations.

environment which allows to run the software under test inside the network simulation tool (see Figure 2.3). Through such mechanisms it is for example possible to execute unmodified applications as part of a network simulation [63].

2.3 Wireless Networks

The perhaps most essential part of every communication system is the physical medium over which the communication takes place. Typical choices are electrical wires, optical fibers and radio waves. While the former two require the installation of cables, the latter one is wireless. The lack of cables makes *wireless networks* very convenient to use, but also more sophisticated from a technical perspective.

This thesis focuses on *wireless local area networks* (WLANs), which are a substitute for local area networks using wired links. The proper definition of a LAN is its size, namely that it spans over an area of up to a few kilometers [99]. What however also characterizes LANs is that their actual implementations include functionality of the physical and data link layer, which both depend on the physical medium used. The nowadays most widely used technology for LANs is *Ethernet* [73].

2.3.1 Challenges of WLANs

The use of radio waves as transmission medium has consequences: While communication over a wire only affects stations which are connected to it, radio waves are only bounded through the distance they travel and obstacles which are in their way. On the one hand, this makes up the big advantage of WLANs, namely that it is so easy to connect to them. On the other hand, this leads to additional challenges which have to be solved by a WLAN implementation.

2.3.1.1 Physical Layer

The task of the physical layer is the conversion between data streams and physical signals being transmitted over the physical medium. By using *modulation*, a number of bits is converted into a carrier signal which can then be transmitted at a specific frequency over the wireless medium.

Frequency regulation

As there are many different services that want to perform wireless transmission, governmental institutions regulate the frequency spectrum by dividing it into licensed frequency bands. The so-called ISM (industry, scientific, medical) frequency bands are license-free, which means that everyone may use them without acquiring a special license. However, this means that different kinds of devices share these frequency bands[1], which leads to interference. In order to minimize the interference, the local regulators make constraints on how these frequencies may be used for data transmissions. An additional difficulty is that these regulations are country-specific and therefore different constraints and channel restrictions apply in each country.

Spread spectrum

One of these constraints for data transmissions is the use of *spread spectrum* techniques [27], which increase the bandwidth of the transmission signal and thereby reduce interference. This bandwidth increase can be obtained through different methods: With *frequency hopping* (FH), transmission channels are switched over time, so that the likeliness of collisions is reduced. This is done according to a pseudorandom sequence which is known to sender and receiver. Another method is *direct-sequence* (DS) spread spectrum, where a so-called *chipping sequence* is multiplied to the transmission signal. This sequence has a higher data rate than the signal to transmit and thereby increases the bandwidth of the resulting signal. A third technique is *orthogonal frequency-division multiplexing* (OFDM), which divides the transmission channel into many small sub-carriers which are used to transmit parallel data streams, each at a low rate. The frequencies of these sub-carriers are placed very close to each other, but in such a way that the parallel transmissions do not interfere with each other, which is the reason for the term orthogonal in OFDM.

Signal propagation

The above techniques provide help in reducing the interference caused by other devices emitting electrical waves, but still can not eliminate it. In addition to the influence of other devices, the transmitted signal itself is also disturbed in various ways by objects which are in its way. This is the reason why the signal quality inside buildings and in cities is worse than in rural environments. Another factor is the signal power, which decreases over distance. A more detailed description of these effects can be found in [92].

High error-rates

The consequence of these effects is that wireless transmissions have to cope with a much higher error-rate than wired transmissions [92], which makes it harder to achieve high data rates. If the transmission power and available bandwidth are fixed, higher data rates can in principle only be achieved by using more sophisticated modulation techniques, which are however more sensitive to noise.

[1]The 2.4 GHz band is for example used by many devices, such as microwave ovens, cordless phones or Bluetooth.

MIMO

One way out of this problem is the use of *multiple-input multiple-output* techniques [27], which use multiple parallel data transmissions at the same frequency. The data is split into multiple low-rate data streams, which are transmitted each over a separate antenna (*spatial multiplexing*). If the receiving antennas receive these transmitted signals in a sufficiently different way, the receiver can distinguish between them and reassemble the original data stream.

2.3.1.2 Medium Access Control

One of the tasks of the data link layer, sometimes represented as medium access control (MAC) sublayer, is to coordinate the medium access. In a wired network, all stations attached to the wire can sense whether a transmission is going on or not. Even if a collision occurs, the sender will notice this and can react on it. The technique used in Ethernet is *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD) [99]. A station which wants to transmit data waits until it senses that the medium is free. If a collision is detected during the transmission, the station waits for a random time and restarts the procedure. This *random backoff* minimizes the risk that the competing stations retry their transmissions at the same time, leading to yet another collision.

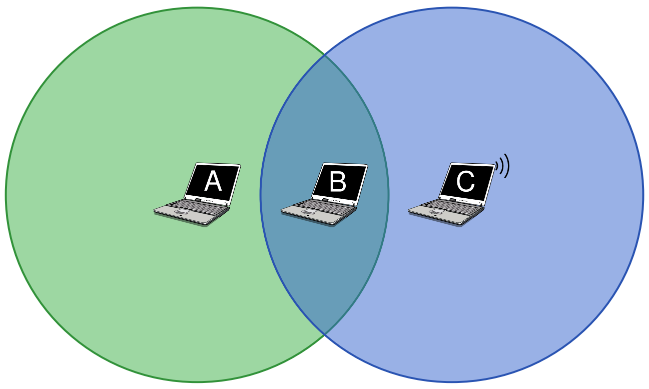
In a wireless network however, the signal does not propagate evenly to all stations and only some stations might be in reception range of each other. As a consequence, a node can only sense ongoing transmissions or collisions taking place at its own position and has no knowledge about the situation at the receiving node. An additional difficulty is that most wireless network hardware cannot listen to the channel while sending, so that a sending node also does not know whether there is a collision at its own position.

Acknowledgment frames

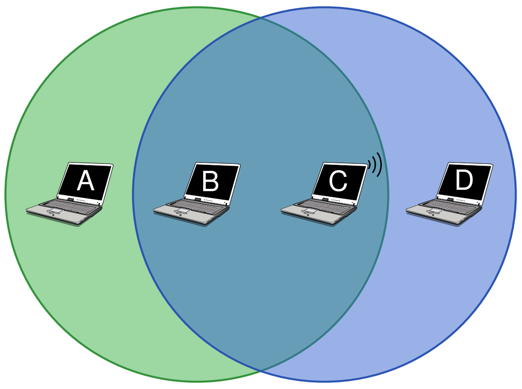
A partial solution to these problems is the use of acknowledgment frames (ACKs). Whenever a station successfully receives data, it transmits an acknowledgment, which is received by the sender of the data. This method was already used in the early ALOHA network [1], which was set up by the University of Hawaii to wirelessly connect computers on different islands. With ALOHA, a station simply starts to transmit data and then waits some time for the acknowledgment of the receiver. If no acknowledgment is received, it waits for a random amount of time and then restarts the transmission.

The hidden-station problem

The use of acknowledgments ensures that a sender knows at some point that its transmission was successful, but does not try to avoid collisions. Due to the problems described above, the CSMA/CD algorithm as used by Ethernet is no solution for wireless networks. In the situation depicted in Figure 2.4(a), A wants to send a frame to B, but cannot sense the ongoing transmission of C. If A now starts the



(a) The hidden-station problem: A sends to B, but disturbs the transmission from C to B.



(b) The exposed-station problem: B refrains from sending to A, although the communication between C and D would not disturb A.

Figure 2.4 The hidden-station and exposed-station problem in wireless communication.

transmission, both signals will interfere and B cannot receive them. This is called the *hidden station problem*. An inverse problem is the *exposed station problem* shown in Figure 2.4(b). Here, B wants to send to A, but senses the transmission of C communicating with D. However C's signal does not reach A, so that a transmission from B to A could take place without problems.

RTS/CTS

Abovementioned problems can be avoided by using a mechanism called *Request to Send / Clear to Send* (RTS/CTS) [58]. If a node wants to send data to another station, it first sends a RTS frame, which the recipient acknowledges with a CTS frame. The idea of this procedure is that stations which could interfere with the recipient receive the CTS frame even though they might not receive the actual transmission and thus know that a transmission is going on. A collision can still occur during the RTS/CTS handshake, but is more unlikely since these two frames have a very short duration.

Virtual carrier-sense

If a station would just receive a RTS frame it would only know that there will be a transmission, but not when it is allowed to transmit again. Therefore the RTS/CTS frames contain a field which indicates the duration of the following transmission. Stations keep track of this using a *network allocation vector* (NAV) which stores how long the medium is occupied. This *virtual carrier-sensing* function can be used in addition to physical carrier-sensing which the radio interface provides. Even without the use of RTS/CTS frames, virtual carrier-sensing can be used by including the duration at the beginning of a transmission. Although only stations which anyhow receive the signal will get this information, they know how long the transmission will last. Thus they do not actively have to sense whether the medium is free and can save energy.

Addressing

As the wireless medium is a broadcast medium, a network with more than two stations needs a way to distinguish the recipient of a message. This is also the case in many wired networks, where address fields in the frame header are used as a solution. A difference is the distinction between different networks. In a wired network, the wire determines which network the station belongs to. But since there is only one wireless medium, this problem has to be solved differently in wireless networks, e.g. with an additional identifier used in frame headers.

Security

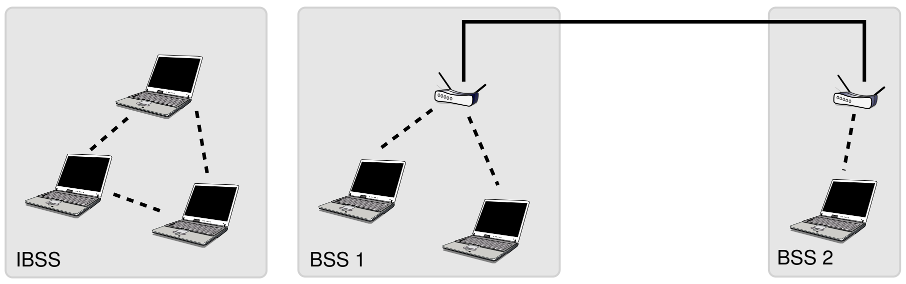
Another issue which follows from this fact is that additional security mechanisms might be necessary. In order to tap on a wired network, it is necessary to access a network cable which is in many cases secure enough. In wireless networks however, it is sufficient to be within proximity of transmitting stations, so that encryption is used in order to protect the network traffic from being read or modified.

2.3.1.3 Compatibility to Ethernet

Since Ethernet is by far the most widespread technology used for local area networks, it might be an additional objective of a wireless LAN technology to integrate into existing Ethernet networks. This basically means that frames have to be exchanged between an Ethernet network and a wireless LAN, which can for example be accomplished by the use of a bridging device. In order to make addressing work between the two types of network, the simplest solution is the use of the same address space, since some form of address translation would be needed otherwise.

2.3.2 IEEE 802.11

Today's most widely used type of WLAN is defined in the IEEE 802.11 standard, which describes both the *Medium Access Control* (MAC) and *physical* (PHY) layer.



(a) A wireless network in ad hoc mode. (b) A wireless network in infrastructure mode with two access points connected through a distribution system.

Figure 2.5 The two types of networks supported by 802.11.

Its initial version has been published in 1997 and provided a data rate of up to 2 Mbps. Later on, task groups developed several amendments to this standard which allow for higher data rates or provide new features. The current version of the complete standard is called 802.11-2007 [45] and includes all amendments which have been published until then. A good overview of the complete 802.11 standard can be found in [39].

The abbreviation *Wi-Fi*, which stands for Wireless Fidelity, is often used as a synonym for 802.11 technology, but is in fact a trademark of the Wi-Fi Alliance [114] which certifies 802.11 compliant devices.

This section describes the basic principles of 802.11 networks, but can of course not go into too much detail. Textbooks such as [30] or the standard itself [45] can be consulted for a more detailed description.

2.3.2.1 Network Types

There are basically two different types of 802.11 networks (see Figure 2.5), commonly referred to as *ad hoc mode* and *infrastructure mode* (sometimes also called *master mode*). In ad hoc mode, the stations send frames directly to each other without any mediator in between, which is convenient to set up because no infrastructure is required. This is the essential difference to the infrastructure mode where all communication takes place over a special station called *access point*. The main purpose of an access point is to act as a bridge to a wired network, but the communication within the wireless network also takes place through the access point.

Just like any other technology, 802.11 brings its own set of terminology: A group of stations which are in reach of each other is called a *basic service set* (BSS) or *independent BSS* (IBSS) in case of ad hoc mode. In infrastructure mode, multiple access points can be connected to each other in order to form one large wireless network, which is then called *extended service set* (ESS). The logic and backbone network used to connect the access points is called *distribution system*. A network is identified by a 32 byte identifier called *service set identifier* (SSID). In case of an extended service set, all access points share the same SSID which is therefore also referred to as ESSID. A single access point is identified by the basic service set identifier (BSSID), which is the access point's MAC address. In case of an IBSS, the BSSID is generated from a random number.

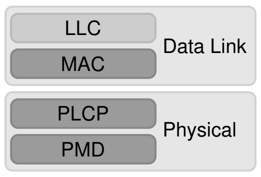


Figure 2.6 802.11 includes the physical layer and the medium access control sublayer.

Amendment	Max. Data Rate	Frequency	Spread Spectrum	MIMO
802.11	2 Mbps	2.4 GHz	FH, DS	no
802.11a	54 Mbps	5 GHz	OFDM	no
802.11b	11 Mbps	2.4 GHz	DS	no
802.11g	54 Mbps	2.4 GHz	OFDM, DS	no
802.11n	600 Mbps	2.4 & 5 GHz	OFDM	yes

Table 2.1 Overview of the physical layers provided by 802.11.

2.3.2.2 Physical Layer

As already mentioned above, 802.11 defines the MAC and PHY layer of WLAN. It not only covers the functionality of both, they are also kept as separate layers in the standard, which has the advantage that they are independent of each other. The PHY layer is even further split up into a *Physical Layer Convergence Procedure* (PLCP) and a *Physical Media Dependent* (PMD) sublayer (see Figure 2.6). Loosely speaking, the task of the PMD sublayer is to bring bits to the antenna. It therefore mainly has to deal with modulation. The PLCP sublayer serves as a binding between the generic MAC and the very specific PMD sublayer.

The advantage of such a layered approach is that it is possible to have different implementations of each layer which can be exchanged independently. This is excessively used in 802.11: There is basically only a single MAC layer, but over the years several physical layers have been developed (see Table 2.1). The split inside the physical layer is also not without reason: Each physical layer provides one PLCP and multiple PMD sublayers, one for each data rate. The different data rates are achieved through different modulation techniques and are used to adapt to the signal quality.

The PLCP adds a preamble, used for synchronization, and an additional header to the beginning of each frame. The PLCP header is used to transmit information which the receiving station needs to decode the following MAC frame: Figure 2.7 shows the PLCP header of the direct sequence PHY layer, which supports data rates of 1 and 2 Mbps: The preamble and PLCP header are always transmitted at 1 Mbps, whereas the MAC frame itself can be transmitted at both speeds. The information which data rate is used is stored in the Signal field of the header. After the receiving station has read the PLCP header, it switches to the corresponding data rate and reads the following MAC frame.

A detailed understanding of the internals of the various PHY layers is not needed for the remainder of this thesis, but is is helpful to have a rough idea of their relation to each other and how they evolved historically. The initial 802.11 standard specified three different physical layers: One using infrared light instead of radio, one using

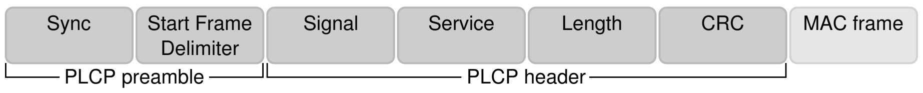


Figure 2.7 PLCP header of the 802.11 direct sequence physical layer.

frequency hopping spread spectrum and another one using direct-sequence spread spectrum. The PHY layer defined in 802.11b [45] is a high speed version of this direct-sequence PHY layer. It is identical for speeds of 1 and 2 Mbps and thus provides backwards compatibility to the original standard. This is the big difference to 802.11a [45], which as the first standard uses OFDM techniques providing higher data rates of up to 54 Mbps. It is incompatible to 802.11 and 802.11b, but this causes no problems on the channel since it uses the 5 GHz band, whereas the older two standards use 2.4 GHz. This lack of incompatibility lead to the development of 802.11g [45], which uses the same techniques as 802.11a, but operates in the 2.4 GHz band. Through some tricks it is still backwards compatible to 802.11b, which uses a different modulation technique.

The newest standard 802.11n [47] also uses OFDM techniques and provides data rates of up to 600 Mbps in the 2.4 GHz or 5 GHz band. This high performance gain is achieved through the use of MIMO techniques, in combination with the use of wider channels and some optimizations in the MAC layer. Although 802.11n incorporates many changes, it is backwards compatible to all previous standards.

2.3.2.3 Medium Access Control

As described above, the medium access in a wireless network requires some extra effort. Following the medium access scheme of Ethernet, the technique used by 802.11 is called *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA) [45]. In contrast to CSMA/CD, the carrier sense is performed both physically and virtually through the use of a network allocation vector maintained by each station. Random backoffs are always performed when the medium becomes free, and not just after a collision occurred. Another difference is that all unicast frames have to be acknowledged by the receiver. An RTS/CTS handshake can be used to avoid the hidden-station problem. However, this takes some extra time and the standard recommends to disable it by default.

The medium access through CSMA/CA happens in a truly distributed fashion, which is why 802.11 also calls it *distributed coordination function* (DCF). For infrastructure networks 802.11 also defines a *point coordination function* (PCF) with which stations do not control medium access on their own, but transfer this task to the access point they are connected to. This mode is optional and not widely implemented [30]. If it is used, PCF and DCF form two consecutive phases: The PCF phase is announced beforehand, so that all stations know that they are not allowed to access the medium via DCF. The access point then polls all stations whether they want to send data. When the PCF phase is over, the medium can be accessed via DCF until the next PCF phase begins.

In order to prioritize the access to the medium, 802.11 uses so-called *interframe spacings* (see Figure 2.8). The shortest one is the *short interframe space* (SIFS),

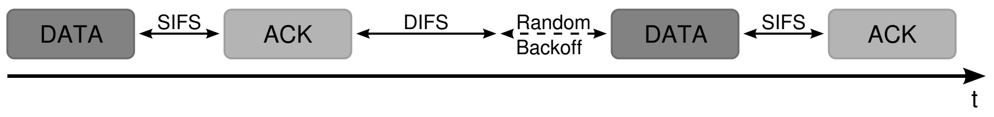


Figure 2.8 The use of interframe spacings during the transmission of data frames.

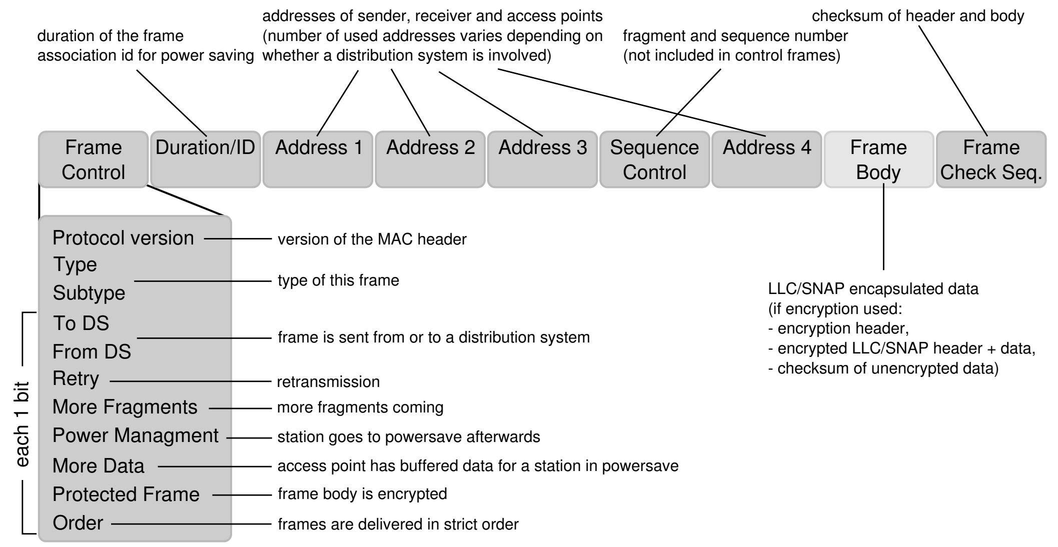

duration of the frame
association id for power saving
addresses of sender, receiver and access points (number of used addresses varies depending on whether a distribution system is involved)
fragment and sequence number (not included in control frames)
checksum of header and body
Frame Control
Duration/ID
Address 1
Address 2
Address 3
Sequence Control
Address 4
Frame Body
Frame Check Seq.
Protocol version
Type
Subtype
To DS
From DS
Retry
More Fragments
Power Managment
More Data
Protected Frame
Order
each 1 bit
version of the MAC header
type of this frame
frame is sent from or to a distribution system
retransmission
more fragments coming
station goes to powersave afterwards
access point has buffered data for a station in powersave
frame body is encrypted
frames are delivered in strict order
LLC/SNAP encapsulated data
(if encryption used:
- encryption header,
- encrypted LLC/SNAP header + data,
- checksum of unencrypted data)

Figure 2.9 The fields of an 802.11 frame and their meanings.

which is the duration after which an ACK or RTS/CTS frame is sent. New data frames may be sent after waiting for the amount of an *PCF interframe space* (PIFS) resp. *DCF interframe space* (DIFS). Since the PIFS is shorter, frames sent in PCF mode are prioritized over frames sent in DCF mode.

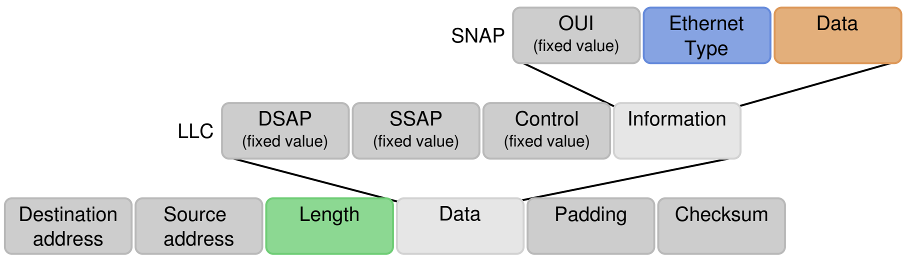
The MAC layer uses three types of frames: data, control and management. Data frames carry payload of higher layer protocols whereas control frames assist the delivery of data frames (e.g. acknowledgment frames). Management frames are used to coordinate the access to a network (e.g. network association or probing for available networks). All three frames types use a similar header, which is sent after the PLCP header. Figure 2.9 explains the 802.11 frame format.

2.3.2.4 Integration into Ethernet Networks

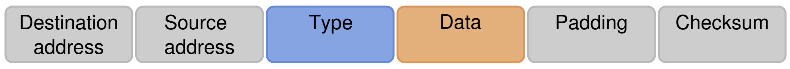
Since today's de facto standard for LANs is Ethernet, one of the goals of 802.11 is to integrate into existing Ethernet networks.

Unfortunately, a number of different versions of Ethernet exist and are used, leading to confusion regarding the frame format: Besides the original Ethernet [73], there is an *Ethernet II* published by the corporations DEC, Intel and Xerox, therefore also called *DIX Ethernet*. Since DIX Ethernet became a widely used technology, IEEE integrated it as 802.3 [46] into its 802 standards family for LANs. However, they made minor modifications to it, including a change of the frame format.

Figure 2.10 shows these two frame formats: Instead of the type field used by DIX Ethernet, IEEE 802.3 stores the length of the frame at this position. The type of the message is indicated through an additional *logical link control* (LLC) header using



(a) IEEE 802.3 (using LLC/SNAP encapsulation)



(b) DIX Ethernet

Figure 2.10 Ethernet as defined by IEEE and DIX Ethernet use different frame formats.

the *subnetwork access protocol* (SNAP), defined in IEEE 802.2 [44]. This LLC/SNAP header is only used to store the Ethernet type, all other values are fixed. Thus, the only additional information carried by the 802.3 format is the length of the frame. This information is signaled by most PHY layers, so that it is not needed as part of the MAC header.

Luckily, it is possible to use both conventions since the data-length of Ethernet is limited to 1500 byte and the type identifiers in use all have values greater than 1500 [43]. Therefore, it is an Ethernet II frame if it contains a value greater than 1500, or an 802.3 frame if it is smaller than or equal to 1500. Today the frame format of DIX Ethernet is most widely used and will simply be called Ethernet in the remainder of this document.

802.11 uses LLC/SNAP encapsulation in the exact same way as 802.3 does, which means that the type field is not included as part of the MAC header, but inside an additional LLC/SNAP header. Another similarity is that 802.11 uses the same 48-bit MAC addresses, which makes it possible to send Ethernet frames into a wireless network and vice versa.

The integration of 802.11 into Ethernet networks takes place through access points, which act as a bridge between a wired and a wireless network. Another point where a conversion between Ethernet and 802.11 frames might be necessary is inside operating systems: Because it shall be possible to use WLAN devices instead of Ethernet devices, some operating systems just pass Ethernet frames to the driver of an 802.11 wireless network card.

One obstacle for the conversion between 802.11 and Ethernet frames is that 802.11 allows frames larger than 1500 byte, namely 2296 byte (plus the 8 byte LLC header). This might lead to problems because 1500 is the magic number for the Ethernet type/length field which is used to decide whether the frame uses 802.3 or Ethernet format. Thus it has to be clear from the context which format is used. This, however, is not a big problem in practice, as most Wi-Fi cards only support to send data with a length of up to 1500 byte[2].

[2]A quick survey on the 802.11 network card drivers included in the Linux kernel showed that only a few of them use a maximum transmission unit larger than 1500 byte.

2.3.3 Testing in Wireless Networks

As described in Section 2.1, the testing and evaluation of communication systems poses challenges. Things get even more involved when wireless networks are used as part of the communication system.

The main reason for this is the complexity of the wireless channel. In a wired network, a node is connected to the network if it is attached to the cable and a rather reliable communication can take place. In a wireless network there is no such clear expression of being connected. While one frame might get through, the next one might get dropped. Already small changes in the environment can lead to errors on the channel, which is why one sometimes speaks of a nondeterministic behavior of the wireless channel. This behavior especially complicates the testing of communication systems, since it is not possible to re-run a test under the exact same conditions, which is often required in order to track down problems. A further consequence is that a system which worked well during tests might fail due to a changed environment, making it necessary to test the system under different conditions.

Another factor is the mobility of nodes. While a station in a wired network is either attached or not, stations may move within the reception range of a wireless network or even switch the network they are associated with. Including such scenarios in the tests of a communication system is obviously difficult, since it requires either many people participating in the test [77], or some automatism [21] to produce such behavior.

The interfaces used by a wireless network can also complicate testing. The service provided by wired networks is usually just sending and receiving data. A wireless network also provides interfaces to choose the network to be associated with, which might be complicated to include into automated tests.

Even though WLANs like 802.11 are a replacement for LANs which can be used through the same interfaces, some behavior of the the wireless network, like an increased error-rate or higher delays, propagates through to implementations residing in higher layers. Thus, these additional testing difficulties do not only affect development of new wireless network technologies, but also systems which just use wireless networks as part of the whole communication system.

2.4 Real Implementations

Section 2.1 has given first insights on the structure of communication systems. This section will provide a more detailed view on this topic and explain how implementations of communication systems are internally structured.

2.4.1 Operating System

Among many, one task of an operating system is the processing of network packets. The central component with regard to this is the *network stack*, which is usually an implementation of the network and transport layer, and thus in most cases the

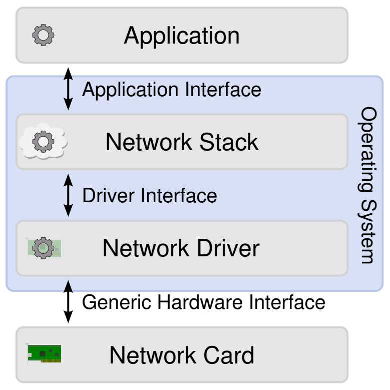


Figure 2.11 The network stack from an operating system's perspective.

TCP/IP protocols [15]. Sometimes, the application layer functionality may also be part of the operating system, as it is the case for example with network filesystem implementations. Besides providing this raw functionality, the OS also provides network-specific interfaces to instances which are not part of the operating system. These are applications which usually interface with the transport layer, and in the other direction network hardware implementing the functionality of the data link and physical layer. A network card is not accessed directly via a standard interface, but through the use of a specific driver program. Figure 2.11 shows the interplay of these components.

2.4.1.1 Network Driver

The driver of a network card runs as a privileged part in the operating system and acts as a bridge between the hardware interfaces of the network card and the software interfaces provided by the operating system. As a consequence, a driver program for every combination of network card and operating system is necessary. While in some cases the driver only acts as a pure interface, it can also take over parts of the functionality which would otherwise be implemented in the network card itself. This includes simple tasks such as the maintenance of state required by the driver, loading the network card's firmware or even complete implementations of the medium access control sublayer (SoftMAC [31]).

2.4.1.2 Driver Interface

In the case of Ethernet network cards, the interface between driver and operating system is rather small: Its main task is to pass Ethernet frames in both directions. Besides that, network cards can be started, stopped, asked for statistics and in some cases additional parameters can be set.

Again, for wireless networks things are more complex. As 802.11 networks are Ethernet compatible, the Ethernet network interface can be used as a foundation. But since 802.11 provides more functionality, this has to be reflected through an extended interface to the network driver which provides additional functionality:

- Extended statistics containing information about the associated network, its signal strength, etc. are provided.
- The channel number, data rate, SSID and BSSID are configurable.
- It can be scanned for networks which are within reception range.
- The mode of the network card can be set: Master, Ad Hoc or Monitor[3].
- Some cards also provide a so-called *Spy* interface, which allows to acquire the signal strength on a per-station basis.

2.4.1.3 Influence on Testing

The way in which an operating system implements these single components certainly has influence on the behavior of a communication system. In addition to this, the operating system also has indirect influence on the complete system through its literal meaning: It operates all components of a system. It initializes the hardware, manages memory and runs its subcomponents and applications. Especially the timing with which the operating system runs these components (the scheduling) has a large influence on the behavior of the complete system.

It is therefore important to include operating systems in the analysis and testing of communication systems, so that their influence on the behavior of a communication system does not get neglected.

2.4.2 Testbeds

As mentioned above, testing of communication systems means that multiple devices are involved in the test. Such a group of devices used for testing and evaluation is called a *testbed*. Some testbeds are set up for just one experiment, but research institutes also build permanent testbeds if they often require a similar environment for testing.

The big disadvantage of testbeds is that actual hardware is required and has to be maintained. This leads to high costs for the acquisition and maintenance, which might be too costly for scenarios requiring large networks. In projects such as PlanetLab [18], research institutions from all over the world work together in order to build a large testbed which is then used by all participants together.

Testbeds for wireless networks are even more demanding. Since the wireless channel depends on the physical deployment of the stations, it is for most scenarios not possible to have all stations together in one room, but they have to be spread over a building or even a larger area (e.g. [95, 2]). Scenarios which involve mobility are even harder to achieve, since it requires additional effort to realize the node movement [21].

[3]Some 802.11 network cards provide a *monitor mode* which captures all received frames, irrespective of SSID and MAC addresses.

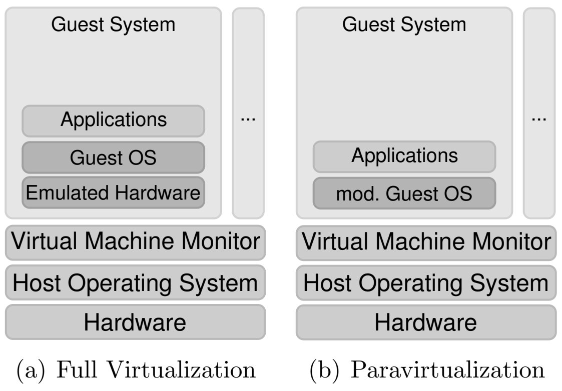


Figure 2.12 Paravirtualization differs from full virtualization in that it requires a modified guest operating system.

Another disadvantage of testbeds is that experiments are in general not repeatable. Through variations of the wireless channel, it is not possible to run a test for multiple times using the same channel conditions. One the one hand, these variations are one of the challenges in wireless network protocols and therefore have to be included in tests. In order to test or debug a wireless network protocol, it can, however, be necessary to repeat a certain test under the same conditions.

2.4.3 Virtualization

A way to avoid the high costs of testbeds is the virtualization of systems. The hardware of a system is emulated, while all software runs as it would on a real system. This means that a regular operating system and application software can be tested without the need of real hardware.

This so-called *full virtualization* requires some performance overhead, because the hardware has to be emulated and certain processor commands of the virtual machines have to be trapped in order to keep them in their emulated environment. In contrast, *paravirtualization* tries to minimize this performance overhead by using modified guest operating systems which are aware of their emulated environment. Through the use of special interfaces to the host system, the emulation of hardware is not necessary, resulting in higher performance. However, this method requires modified guest systems and cannot be used to run arbitrary systems. Figure 2.12 shows the difference between these two techniques. Both methods have in common that they require software on the host system which actually runs the virtualized systems. This software is called *virtual machine monitor* [32].

Besides the savings in hardware needed for such a setup, virtualization can also open up new possibilities: Since the operating system is usually the lowest level of software running, it is rather complicated to debug a networking implementation which is part of the operating system. With the indirection layer of virtualization in between, it is however possible to tap into the operating system in order to debug the operating system [57, 59]. This can be especially useful for the development of communication systems, since many network protocols are part of the operating system.

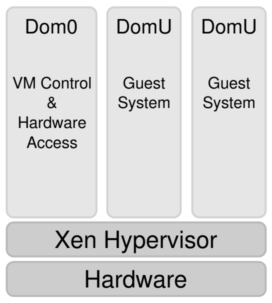


Figure 2.13 The Xen virtualization system uses a hypervisor which runs directly on hardware.

2.4.3.1 The Xen Hypervisor

A popular virtualization system is Xen [118], which differs from systems as depicted in Figure 2.12 in that it does not use a virtual machine monitor running on the host system. Instead, the *hypervisor* fulfilling these tasks is running directly on hardware. All systems running on the hypervisor are called *domains*, whereas the domain 0 (Dom0) has a special functionality: It is the first system started together with the hypervisor and is used to control the other systems. It is also the only system allowed to access hardware directly. The other domains are unprivileged (DomU) and are only allowed to access hardware indirectly through Dom0. Figure 2.13 shows the structure of a system using Xen.

Xen supports both paravirtualization (PVM) and full virtualization, which is called hardware virtualization mode (HVM) in Xen terms. For HVM systems, it makes use of processor extensions, which support the required context switching, but are only included in recent processors. More information on Xen can be found in [83, 17].

2.4.3.2 Testing of Communication Systems

As many virtualization solutions also offer the ability of network connectivity through emulated network cards, such systems can be used for the evaluation and testing of communication systems. This networking functionality is usually realized by an emulated Ethernet device in the guest system, which passes sent and received frames to a virtual network device in the host system. By connecting multiple of these interfaces in the host system, it is then possible to give the virtualized systems the impression of a common network they are connected to.

The main constraint regarding the evaluation of communication systems is the emulated hardware. If the virtual network devices that are provided by the virtualization tool are sufficient for the system under test, virtualization can be a good tool as it allows to test multiple hosts of the communication system by using a single real machine. Since everything from the operating system upwards is a real implementation, a good level of detail is achieved.

In such cases, virtualization reduces the amount of hardware needed, but scalability is still a big issue as only a limited number of virtual machines can be run on one real machine.

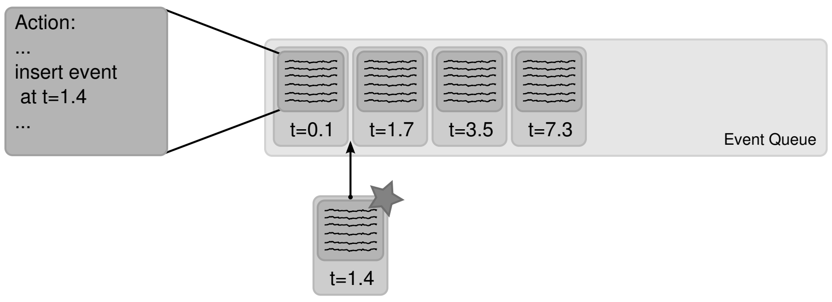


Figure 2.14 The principle of the event queue used in discrete event simulation.

2.5 Network Simulation

One way to evaluate the behavior of large communication systems are network simulations: Instead of using a real implementation, only the behavior of a hypothetical system is modeled. A widely used type of simulation is *discrete event simulation* [28] (DES).

The general principle of discrete event simulation is that all actions which would happen in a real system (i.e. everything that changes the state of the system) are modeled as events which occur at a specific time. Thus, what makes up an *event* is a certain action that is performed and an associated time at which that action is executed. Since every event is executed at a specific single point in time, the events are discrete. Some events are *scheduled* initially and later on events may in turn *schedule* new events and so forth.

Since this general concept is used often, complete tools exist which can be used for the development of discrete event simulations. While tools as ns-3 [38] are specifically tailored towards network simulation, others as OMNeT++ [107] are kept generic and can be used for simulation of other fields as well.

This concept is commonly implemented using an *event queue*, a queue which contains all pending events ordered by their associated time. The simulation program then removes the first event of this queue, executes the program code associated with the event and then proceeds to the next event. This process is depicted in Figure 2.14.

The time inside a discrete event simulation is completely decoupled from the real time: The time inside the simulation is always equal to the time of the event currently being executed and one therefore speaks of *simulation time*. The external time of the real world is called *wall clock time*.

In the context of network simulation, the stations of a network are called *simulation nodes*. The other kind of simulation objects are channels, which are used to connect the nodes to each other. As the functionality of simulation nodes resembles the one of real systems, they are often structured as separate modules, each representing a single layer of the network stack.

Due to the nature of DES, everything that happens in the node and channel objects has to be modeled through events. For example, a channel sends a packet by scheduling an event for the receiving node at the current time plus the channel delay. Similarly, events can be used inside nodes to model timers, as needed for example for a random backoff channel access or the retransmission of packets.

2.5.1 Evaluation of Communication Systems

Discrete event simulation can be a good help for the evaluation of communication systems, as it allows to model a system with an arbitrary level of abstraction. If for example a detailed model of the physical layer is not needed, one can simply leave it out and can make two nodes communicate with a more abstract view of the channel, e.g. by just modeling packet loss and delay. Another advantage is that there are simulation models already available for many simulation tools, so that it is not necessary to implement everything from scratch.

The level of detail used in simulation models not only influences the programming effort, but also impacts the performance of later simulation runs in terms of runtime and memory [36]. Another limiting factor of a network simulation is the number of nodes being part of the network [113], but as long as there is sufficient memory, it is possible to run simulations of arbitrary size. This happens at the cost of runtime: Although the time of a simulation is decoupled from the real world and the simulation can run as slow as necessary, at some point it simply takes too long to wait for the results.

Still, in many cases network simulation scales much better than emulation using real systems. Even if simulation using a single computer is not possible any more, parallelization techniques can be applied in order to run the simulation in a distributed fashion across multiple machines [29]. And in some cases simulations are the only way to evaluate a communication system, since network sizes are too large for the use of real systems (e.g. for the evaluation of peer-to-peer networks).

Another big advantage of network simulation is the level of control. Since the system under test is completely synthetic, it is possible to change every detail of the system in any desired way. This includes that a simulation can be repeated arbitrarily often under the same conditions, which is often not possible with real communication systems. The collection of results is also very easy, since all details of the system can be monitored and the data is available at one place.

The use of network simulation also has its drawbacks. One such point are the abstractions which simulations inevitably make. In many cases, the abstractions made can have an effect on the system under test, so that results using a real system would differ. Another disadvantage is the fact that a double development effort is required in many cases. First the simulation models are developed, and after the system has been evaluated by means of network simulation, the real implementation of the system has to be developed and tested again. In some cases it is possible to use parts of the source code for both network simulation and the real system [11], but often different environments, interfaces and programming techniques prohibit such a procedure.

2.5.2 Simulation as Network Emulator

The general concept of network emulation has already been described in Section 2.2. This section explains the use of network simulations for emulation scenarios in more detail.

Network emulation in this context means that a network simulation is run together with one or more implementations of real systems. Technically this means that these systems have to be connected in some way. As both simulation nodes and real implementations are usually structured using the same communication layers, it is possible to connect them at such a layer-boundary [26]. In this way it is possible to use the simulation for the lower layers and the real system for the upper layers of a communication stack, or the other way round.

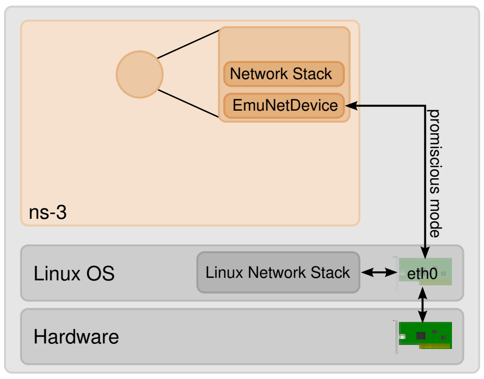
The easiest way to realize such an integration is the use of existing interfaces, which are on the real system's side the network interfaces provided by the operating system. Inside network simulations, similar interfaces exist between the different modules making up the communication stack. Since such simulation nodes transfer a part of their functionality to the real implementation they are connected to, they leave out a part of the functionality in the simulation and are therefore sometimes called *ghost nodes* or *gateway nodes*.

A part of the flexibility which network simulations offer can also be used in emulation scenarios: The level of detail used in the network layers implemented through the simulation can still be varied, which allows to use the complex implementations of real systems with an abstract network model in the network simulation. Another degree of freedom is whether all nodes of the simulation are used as emulation gateways or if a part of them still functions as regular simulation nodes. Such a combination can be used to test a few real implementations in a large network of which most nodes are simulated.

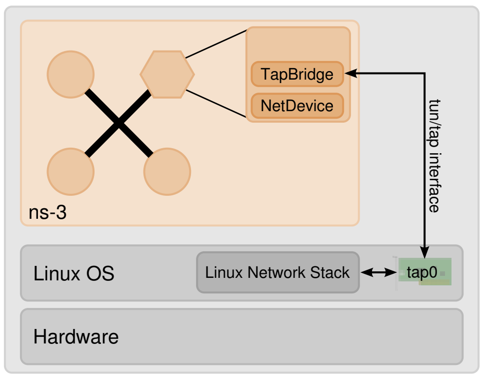
The abstraction of network simulations has its price: Simulations usually do not use a real representation of network packets, but only store the information which is relevant to drive the simulation. While this eases network simulations and leads to a performance gain, it is an obstacle for the integration with real systems: When network packets are exchanged at gateway nodes, they have to be translated into the respective representation, which can lead to an additional development effort.

Another difficulty in the realization of network emulation are the different time concepts. Usually, a network simulation runs independently with its own notion of time. If packets are exchanged with external systems, there is a mismatch in the time concepts. Although no direct time information, but only network packets, are exchanged, a time mismatch becomes visible through the delay of packets or expiring timers.

One solution to solve this time-mismatch is to run the simulation in real-time [26] which means that events have to be executed at exact points in real time. The disadvantage of this solution is that it only works as long as the simulation is real-time capable. If the load of the network simulation is too high, events cannot be processed fast enough and a drift in time occurs. If speeding up the simulation is no option, the only way out of this problem is to slow down the real implementations. This can be achieved through the use of virtual machines, with which it is also possible to change the time perception of the systems running in them. Thus it is possible to let the virtualized system perceive time as if no slow-down would happen. While some approaches just slow down the execution using a fixed factor [24], more sophisticated approaches dynamically adapt to the load [33] or pause both network simulation and real implementations depending on which one is slower [112].



(a) EmuNetDevice



(b) TapBridge

Figure 2.15 ns-3 supports two different methods to connect to real systems.

A modern network simulation tool which is well suited for network emulation is *ns-3* [103, 38]. It contains detailed simulation models for all layers of a network stack and the internal structure of these components very much resembles the one of real communication systems. Since it uses network-byte order as its internal representation of network packets [37], no conversion is necessary if packets are exchanged with real systems. ns-3 includes a scheduler which allows to run the simulation in real-time and provides two different ways of connecting to real systems: The *EmuNetDevice* can be used to run the network stack of a simulation node on a real network device, whereas the *TapBridge* is used to integrate the network stack of a real implementation into a network simulated in ns-3. Figure 2.15 shows these two concepts. Technically, the difference between the two approaches is that the EmuNetDevice connects to an already existing interface, whereas the TapBridge creates a new tap-interface[4].

[4]Tap-interfaces are a mechanism of Linux which allows applications to create virtual network interfaces. All network packets which are sent to such an interface are forwarded to the application. Likewise, the application can insert packets which are then received by the interface

3 Conceptual Design

This chapter explains the wireless network emulation concept which this thesis proposes. To begin with, Section 3.1 defines the goals to achieve. After that, Section 3.2 explains the general approach and Section 3.3 gives more detailed information on its single components.

The focus of this chapter is on the concept itself and does not deal with implementational issues. Details about the implementation provided by this thesis follow in Chapter 4.

3.1 Objective

The goal of this thesis is to develop a concept and a tool to ease the development process of network protocols and applications which make use of wireless local area networks. As described in Chapter 2, the process of testing and evaluating implementations using wireless networks can be difficult and cumbersome. Yet, there are many types of software which make use of wireless networks, whose developers would profit from a simplification of the development process:

- Handover between wireless networks (e.g. [119]).
- Mobile ad hoc network protocols using link-layer metrics (e.g. [69]).
- Localization using wireless networks (e.g. [65]).
- Tools used for the management of wireless network connections.
- Any software using a wireless network for communication.

Although software of the last category does not access any specific features of wireless networks, it may act differently in a wireless environment, for which reason an evaluation in wireless networks might also be necessary in this case. The reason is that connection properties of wireless networks, such as packet loss or delay, can differ widely from those of wired networks. In order to evaluate whether these effects have any influence on a communication protocol, a test in a wired network would not be very helpful. But for software directly accessing features of a wireless network, such as incorporating the wireless signal strength for routing decisions [69], an evaluation using wireless networks is essential.

In either case, an evaluation using real wireless networks is cumbersome, and sometimes even impractical (e.g. for large networks including mobility). In such cases, the use of simulations is possible, but does in most cases not allow to test the real software used for deployment later on. The level of abstraction may also be too high if pure simulations are used. A combination of both approaches, in form of network emulation tailored towards the evaluation of wireless networks, is therefore the objective of this thesis.

In order to achieve the overall objective, the concept which is presented in the following sections was developed with the following design goals in mind:

- Experiments should be **repeatable** under the same conditions. This is for example necessary for debugging of software or in order to experiment with different parameters of an implementation.
- The wireless environment should be **realistic** and fully **controllable**, which includes effects such like different channel conditions or mobility. Realism is absolutely vital for the evaluation of communication systems and is best achieved in real environments. Yet, control over the wireless environment is best accomplished in simulations.
- **Real implementations** should be usable to the greatest extent, so that there are not too many restrictions regarding the types of software for which such a system can be used. If for example only applications and no complete operating systems could be used, many network protocol implementations would be excluded through this.
- The solution should be **scalable** in two dimensions: First of all, the number of real implementations used in the setup should be easy to increase. In addition to that, it should be possible to incorporate simulated nodes for **hybrid emulation** scenarios, which allow the use of only a few real implementation in a nonetheless large network.
- The real implementations should be able to use wireless networks with their **full functionality**. If some of the features or interfaces provided by wireless networks are not available, implementations using them are excluded. This is a requirement for the **transparency** of the system, meaning that real implementations should not be able to perceive that they are not using a real wireless network.
- The overall system should be **cheap**, **simple** and **easy to use**. It should therefore not require any special hardware, lots of resources or complex software setups.

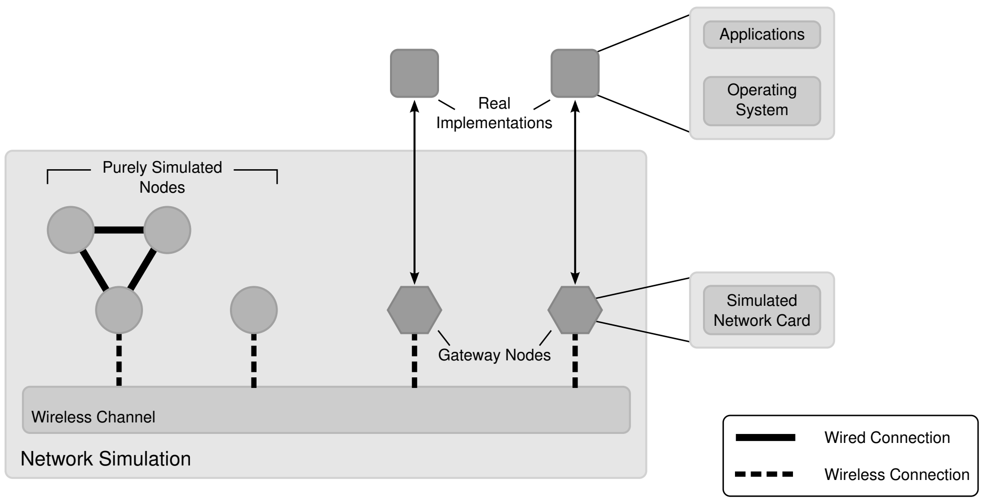


Figure 3.1 An overview of the complete system.

3.2 Approach

As already indicated in Chapter 2, some of the design goals listed above are characteristic for network simulations, some for real systems: The ability to repeat experiments, full control, scalability, low cost and ease of use are properties which are at least in a wireless context more specific for network simulations. The use of real implementations, realistic environments and full-featured wireless networks are however more characteristic for evaluation using complete real systems (as for example in testbeds).

Using discrete event network simulations together with real systems helps to obtain the positive properties of both approaches. Depending on how each methodology is used in such a combined setup, their positive and negative characteristics become apparent in the overall system.

3.2.1 The Overall System

Figure 3.1 shows the complete system as proposed by this thesis. The basis is formed by a network simulation with a common wireless channel, to which two sorts of nodes are attached. Some nodes are *purely simulated*, which means that they are regular nodes in the network simulation. The other sort of nodes are *emulated nodes*, each representing a real implementation attached to them.

The purely simulated nodes are optional and can be used to extend the emulated scenario if not all functionality shall be implemented through real implementations. They can be used for the simulation of all network protocols for which models exist in the used network simulation tool. Thus, they could fulfill the same tasks as the emulated nodes or also different functionality.

An emulated node consists of two parts: A *gateway node* in the simulation is connected to the wireless channel, but only includes functionality of the physical and data link layer. The functionality of the upper network layers is covered by a real implementation running outside the simulation. From a technical perspective, these are two different systems, but regarding the stations in a network (or nodes in a simulation), they form a single system. How exactly this integration inside the emulated nodes is realized, is subject of Section 3.3

The system can be used in two different ways: If only emulated nodes are used, it can serve as a replacement for a wireless testbed. Real systems run the software under test, but instead of accessing real network cards, the ones inside the network simulation are used. Thus all software can be used in its regular way, the only difference being that the evaluation of the system takes place with the simulated wireless network instead of a real one. Through the advantages of the simulated wireless channel, it is possible to re-run the system under the exact same conditions or model effects like varying channel conditions, mobility, etc. However, it is required for each emulated node to run a real operating system and the software under test, which limits the scalability of such setups.

The second way of using the system is to run a hybrid emulation, which uses both emulated and purely simulated nodes. While the emulated nodes are used to evaluate the implementation under test, the simulated nodes can be used to extend the network in terms of size or functionality. For instance, a large wireless ad hoc network can be evaluated by implementing most of its nodes as purely simulated nodes, with just a few real implementations used as emulated nodes. Through this it is possible to evaluate the behavior of a real implementations in a large network, which nevertheless requires only few resources. The purely simulated nodes can also be used to implement functionality which is different from the one provided by emulated nodes. For example, they can be used to model a backbone network, or some service being used by the emulated nodes.

3.2.2 The Split Network Stack

As depicted in Figure 3.2, the functional division of emulated nodes is achieved using the interface between data link and network layer[1]. This means that the gateway node inside the network simulation has to take care of all functionality belonging to the physical and data link layer, whereas an external real implementation runs all components that cover functionality from the network layer upwards.

The split between these two layers resembles the division of real systems into hardware and software. The physical and data link layer are usually the task of the network card, whereas the remaining layers are implemented as part of the operating system or application software. Hence an interface between those layers always exists, which makes a split at this position a suitable choice. Another positive property is the fact that all functionality which is covered by the real implementations is pure software, so that there is no direct involvement of specific hardware in the overall setup.

[1]If one considers the data link layer as two separate sublayers for medium access control and logical link control, one could also say that the interface between these two sublayers is used. However, in practice such a clear distinction is not possible: 802.11 network cards for example only implement the PHY and MAC layer, but replace Ethernet headers through a different LLC header prior to transmission.

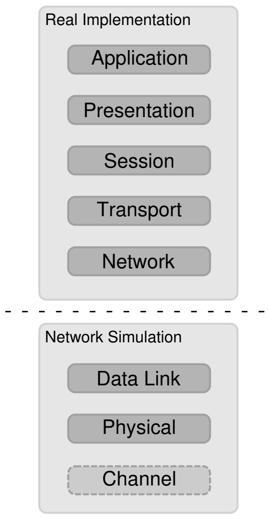


Figure 3.2 The network stack is split up at the interface between data link and network layer.

Furthermore, the decision where to make this split in the network stack has an influence on the properties of the complete system: The advantages and drawbacks of the network simulation mainly affect those layers covered by the simulation and vice versa for the real implementations. In this case, this means that the problematic wireless network is under full control of the network simulation, while the implementation under test can nevertheless be run in a realistic environment.

3.2.3 Design Alternatives

A split at a different layer-boundary would also be possible, but would have other properties than the chosen solution.

Simulating only the channel

In principle, it is also possible to simulate only the wireless channel itself. This means that the physical layer is part of the real implementations and thus hardware in form of network cards is required. As the interface network cards used to access the wireless channel are antennas, this would involve further difficulties, as the network simulation would have to interface with the antenna ports. Nevertheless, such an approach would be possible if some extra hardware were used to connect to the network simulation. Projects like the CMU Wireless Emulator [55] actually go this way, but just as a pure network emulator without making use of a network simulation.

Splitting between physical and data link layer

The next possibility would be to split between physical and data link layer, as it is done with wireless network cards that implement the medium access control as part of the driver (SoftMAC). Thus, such a solution does not require any network hardware as part of the real implementations. Having the MAC functionality as part of the real implementation might seem to be an advantage at first sight, since more control is retained in the real systems. However, the focus here is on the evaluation of systems that just *use* standard wireless networks, so that no modifications of the MAC functionality would be necessary anyway. Another drawback is the high timing-accuracy that such an interface between simulation and real implementation would require: For instance, the interframe-spacings of 802.11 are in the range of tens of µs, so that the MAC implementation residing in the real implementation would also have to communicate with the PHY layer in the network simulation at such a high speed.

Splitting at higher layers

Splitting at a layer-boundary above the network layer would also be possible, but is unsuitable for a number of reasons: In practice, the network and transport layer are often implemented together as part of the operating system, for which reason there is no clear interface between them. In case of the TCP/IP protocols, checksums used in the transport layer rely on data of the network layer, so that it is even impossible to separate them. Splitting above the transport layer is not possible at all, since the session and presentation layer are either left out in practice or are a specific part of the application layer. So the only option would be a split between the application and transport layer, which has the drawback that network protocol implementations which are part of the network or transport layer cannot be used as part of the real implementation. Besides this, the special interfaces which wireless networks provide logically belong to the MAC layer, so that their use together with an interface of a higher layer would not be useful. For instance, joining a new wireless network would not make much sense if it is not possible to change the network address.

3.3 Emulated Nodes

Each emulated node is implemented as two parts: As a gateway node in the network simulation and as a real system running outside the simulation. From a functional point of view, they are the same system, each implementing a part of the network stack. Thus, this split has to be transparent for both systems, so that they perceive the emulated node as a single system.

This functional integration is achieved by connecting the two systems in such a way that the respective missing components are replaced through the other system. The interface utilized for this is the interface between data link and network layer. Thus, the data exchanged between them are data frames sent to or received from the wireless network, plus some control information of the wireless network.

Technical requirements are the appropriate interfaces inside the network simulation, plus the real systems, which have to be connected to each other. These three building blocks are described in the following.

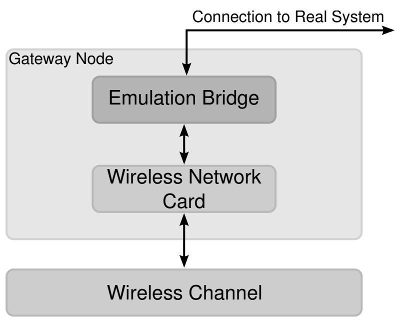


Figure 3.3 A bridging module is used for the integration of emulated nodes into the network simulation.

3.3.1 Network Simulation

Inside the network simulation, each emulated node is realized through a special gateway node which serves as a gateway to the real implementation. Since the network simulation only implements the physical and data link layer of the emulated nodes, these gateway nodes only consist of the simulation's equivalent of a wireless network card which is connected to a wireless channel.

The internal structure of a simulation node often resembles the one of real systems (e.g. [38, 88, 110]). Just as with real implementations, this has the advantage that different modules can be used in different combinations. For instance, a module implementing the TCP/IP protocols might be used on top of a wireless network card module or a wired equivalent. In order to connect these modules, some form of interface is provided by the network simulation.

Such an interface can be used to connect gateway nodes to real implementations. Instead of a module implementing network layer functionality, a new bridge module is installed in the gateway nodes. This bridge has to communicate via some mean with the real implementation, which means that data frames and additional control information have to be exchanged. Figure 3.3 depicts such a setup.

An additional issue which has to be solved is the time-mismatch between a discrete event simulation and the real world. This is not specific to wireless network emulation, and already existing solutions like the use of a real-time scheduler or synchronization mechanisms (see Section 2.5.2) can be used.

3.3.2 Real System

Real systems are used to run the software under test and everything else that is needed to cover the functionality of the network stack from the network layer upwards. Since the goal of using real systems is to provide a genuine environment to the networking software being evaluated, this is in most cases a regular unmodified operating system and some applications.

Although an operating system and applications suffice to cover the functionality of an emulated node, this is not sufficient to build the actual system: An operating system requires a computer to run on. In principle it is possible to use one physical machine per emulated node, but this rapidly leads to scalability issues, as many resources in the form of hardware and maintenance are required.

In order to mitigate these effects, virtual machines (see Section 2.4.3) can be utilized. One virtual machine is used for each emulated node of the setup, running an operating system, applications and among them the software under test. Still, enough computational resources have to be provided in order to run the virtual machines, but at least multiple emulated nodes can be run per physical machine. How many virtual machines can be run on the same machine depends on the hardware resources provided and the complexity required by the system under test. As [8] explains, the memory footprint of simple communication systems is quite low, so that it is possible to run dozens of virtual machines in parallel. Thus, in some cases a single physical machine can be enough to run both the network simulation and all virtual machines.

Regardless whether real systems or virtual machines are used, the essential part is to integrate them into the overall emulation environment, which means connecting them to the gateway nodes of the network simulation. The real implementation is responsible for all layers from the network layer upwards and hence the interface between data link and network layer is utilized for this. Although these two layers are adjacent from a network's point of view, in real systems they are implemented as two components not interfacing directly with each other. The physical and data link layer are implemented as a network card, which is a piece of hardware being connected to a hardware bus. The network layer is part of the operating system's network stack, which is pure software. The network card's driver program, which is also part of the operating system, is used to bridge this gap between network card and network stack. Thus from an implementational point of view, there are two different interfaces which could be utilized to connect to the network simulation.

The first option is to leave the operating system unmodified and interface with an already existing network driver. This would mean that a hardware bus has to be used, which requires building additional hardware which behaves like the wireless network card this driver is written for. If virtual machines are used, such a virtual network card would not have to be built in hardware, but could be implemented in software[2]. Still, in both cases this would be a fairly complex task.

The alternative, and this is what this thesis proposes, is to utilize the interface between the driver and network stack inside the operating system. In practice, this means that a driver program has to be written which behaves like the driver program of a real network card, but connects to the network simulation instead of real hardware. The advantage of this approach is that such a task is less complex than to use the interfaces of real hardware. The drawback is that such driver programs are specific to the operating system used, and it is therefore required to write such software for every operating system being used.

Figure 3.4 shows the structure of this concept. From the operating system's point of view, there is just a network driver which behaves like the one of a real wireless network device. The fact that this driver is connected to the network simulation and not to a real network card is hidden from the system.

[2]For instance the QEMU machine emulator [10] implements network cards in software.

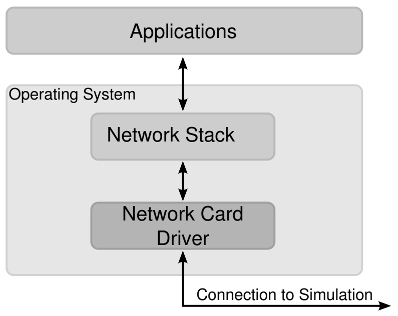


Figure 3.4 A special network card driver is used for the integration of an emulated wireless network into real systems.

In order to make this fully transparent for the implementation under test, it is important that the driver of the wireless network card behaves exactly like the one of a real wireless card. This includes the behavior of the wireless network device and the wireless channel, which are part of the simulation, but also the interfaces used by the driver. If important features which real wireless network cards implement are not supported by the driver, the operating system and hence also applications could notice this through missing interfaces or atypical behavior. There is, however, some margin for this, since the features and the behavior of real wireless network cards vary.

3.3.3 The Connection

The last two sections have explained at which points the integration inside the network simulation and the real systems takes place. It has not yet been covered how these two systems should be connected to allow for the exchange of network frames and control information.

As the abovementioned concept can be applied to different operating systems and network simulations, the connection between them should also provide this flexibility, so that different types of systems can be used together. In addition to this, there are three different methods to run a real system: Native on a dedicated physical machine, encapsulated in virtual machines running on separate systems, or in virtual machines run together with the network simulation on the same system.

Another requirement for a connection between these two systems is the latency induced through it. In order to keep the overall system transparent for the implementation under test, it is important that the delay of the communication between the network card in the virtual machine and the driver in the operating system is in the same order of magnitude as the delay between a real network card and its driver.

Shared memory

If virtual machines are used, a very efficient way to transfer network data to and from the host system is the use of shared memory [54]. One can either use a memory page shared by guest and host system or a special mechanism to copy the data

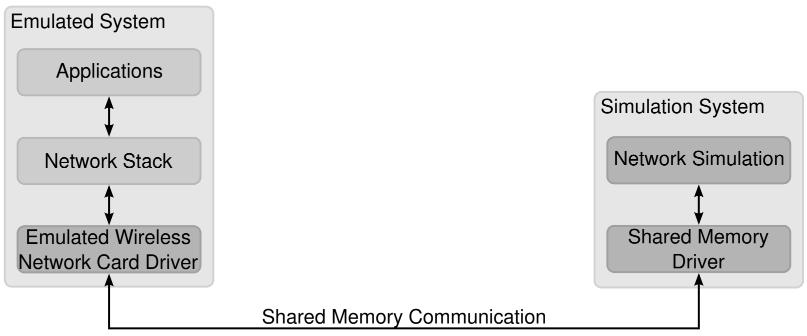


Figure 3.5 Communication of the network simulation and device drivers inside virtual machines via shared memory.

directly between the systems[3]. Although this type of communication is very fast, it is restricted to the use of virtual machines. Furthermore, it requires additional support from both sides: The network driver inside the virtual machine has to use the shared memory for communication, while the network simulation also requires a mechanism to use the shared memory An additional driver in the host operating system might therefore be needed, since such operations are usually only allowed for software running as part of the operating system. This driver then has to communicate with the network simulation, which could happen via some operating system mechanism or a network if the network simulation is running on a different host than the virtual machines. This communication between the network simulation and the shared memory driver in turn creates an additional delay in the communication which mitigates the original advantage of shared memory. Figure 3.5 gives an overview of such a setup.

Network communication

Besides its complexity, such a shared memory solution has the drawback that it has to be implemented for a specific virtual machine technology and therefore does not allow the use of arbitrary virtual machines or a direct execution of real implementations without the use of virtual machines. A much more flexible solution is the use of network communication as a means of exchanging data between network simulation and device driver. A network connection has the advantage that it can be used on a single machine, but also in distributed settings with different physical machines. Furthermore, operating systems provide ready interfaces which allow to access the network functionality from the operating system and regular applications, so that both the device driver and the network simulation can utilize these existing interfaces. As Figure 3.6 shows, the architecture of such a system can make use of the existing networking functionality and does not require any additional components.

The approach of using network communication can also be used if virtual machines are used to run the real implementations. In this case, the device driver is part of the operating system running inside the virtual machine and a network communication between the guest system and the host system has to be established. As such functionality is a regular requirement, virtual machines provide network cards which

[3]For example, the Xen hypervisor allows to copy data directly from and to memory locations used by the Dom0 and DomU systems.

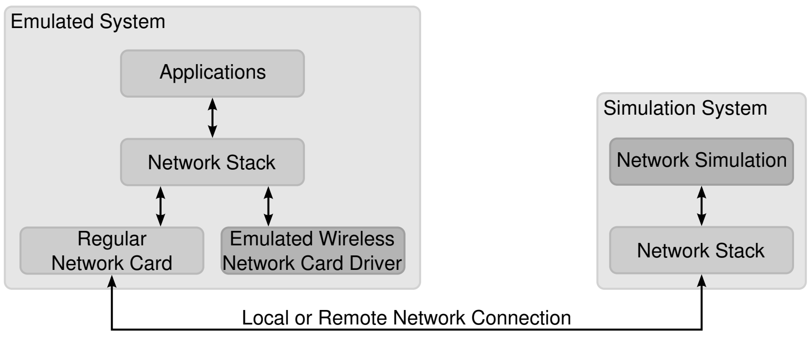


Figure 3.6 Communication of the network simulation and device drivers via a network connection.

can be utilized to establish a network communication to the simulation running as part of the host system or on a remote host.

In comparison, both approaches have their advantages and drawbacks. The exchange of data using a network connection is much more flexible and easier to implement. A possible advantage of using shared memory is a fast communication between the device driver inside the virtual machine and the host system. However, an additional communication step between the host operating system and the network simulation is still necessary, which requires additional implementation efforts and may also cause an additional delay.

4 Implementation

In addition to the concept presented in Chapter 3, this thesis presents an exemplary implementation of this concept. The goal of this chapter is to introduce this implementation and thereby explain how abovementioned concept can be implemented.

Figure 4.1 shows the three components which have to be built in order to implement this concept: A bridge device for a network simulation, a wireless network card driver for an operating system and a common messaging mechanism used for data exchange between them. As all three components depend on the type of the wireless network which is used, it is much easier if these components are developed for a specific wireless network technology. This implementation is therefore restricted to wireless local area networks using the nowadays prevalent IEEE 802.11 [45] standard.

These three components which have to be developed do not stand completely on their own, but have to be integrated into existing infrastructure. The bridge device makes use of a network simulation tool providing an 802.11 wireless network model and the network card driver has to be developed for an operating system which provides interfaces for wireless network cards. Finally, the mechanism used to exchange messages between those two components also relies on existing infrastructure and has to be integrated in both the network simulation and the operating system hosting the network card driver.

Although any implementation of abovementioned concept makes use of these existing systems, it can still be implemented in such a way that some flexibility regarding the choice of these systems is preserved. This flexibility is especially important for the network driver's side, so that different operating systems can be used in one combined evaluation setup. Being able to choose from different network simulation tools or message exchange mechanisms is less important, as only one of them is used at a time.

This implementation uses ns-3 [103] as network simulation tool and Linux [102] as operating system for the real systems. However, the message exchange mechanism is independent of these specific systems, so that it is possible to make use of different

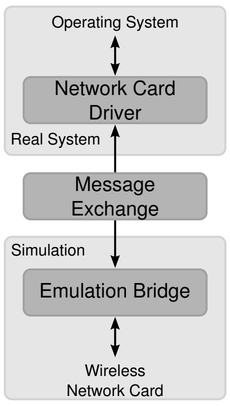


Figure 4.1 The three components which have to be implemented.

network simulation tools or develop network card drivers for other operating systems as well.

4.1 Functionality

Although the simulation and the driver are two independent software components, they belong closely together: The network driver in the real system and the network card in the simulation have to work together as if they were driver and hardware of a physical system. This means that they build a single functional unit in which the driver acts as a software representation of the network card. As such, they share the same features and tasks, which have to be implemented across both components.

Inside the network simulation, the required functionality could be added directly as part of the simulated network card. In order to keep the emulation-specific aspects separated from it, an intermediate emulation bridge using the regular interfaces of the network card is added.

The emulation bridge and the network card driver together have to provide the same functionality which a regular wireless network card provides. Some functionality is already covered by the wireless network card in the network simulation and can be accessed by passing through the according commands. Additional functionality, which is not provided by the simulated network card, has to be implemented in either the emulation bridge or the network driver.

From the perspective of the operating system, the functionality which has to be provided roughly comprises four tasks:

- Sending and receiving of data frames over the wireless channel.
- Setting parameters such as the associated network, channel, bitrate, etc.
- Scanning (probing) for available networks.
- Retrieving statistics from the wireless network card.

Send and receive

To send and receive data is clearly the main task of every network card. This is the reason why both the 802.11 network card of ns-3 and the Linux network stack provide interfaces which allow to send and receive data frames. It therefore basically suffices to connect these interfaces with each other, so that frames sent by the network stack are handed over to the simulated network card and vice versa.

In order to closely match the behavior of real 802.11 network cards, the implementation uses a virtual buffer in the sending direction: A real network card has a buffer in which it enqueues any frames which have to be transmitted. If this buffer is full, a wireless network card driver for Linux can temporarily shut down the network interface so that the network stack does not hand over new packets to send [19]. In order to resemble this behavior with the emulated network card driver, it makes use of a *virtual buffer* which is just a counter indicating how many bytes are enqueued. If this counter reaches a certain threshold, the virtual buffer is full and no new packets can be enqueued.

Another feature which many real 802.11 network cards provide is the so-called monitor mode. In regular operation, the network card driver only forwards received data frames which are destined for the own MAC address. In monitor mode, all received frames irrespective of their type and recipient address are forwarded to the operating system. Furthermore not only the payload, but also the 802.11 headers of the frames are included. As this is an useful feature for the observation of 802.11 networks, this implementation supports it as well.

Parameters

In order to access an 802.11 network, a network card requires a number of parameters: The channel number, the mode of the network (infrastructure or ad hoc) and the SSID or BSSID are used to determine a specific network. Additional parameters supported by the network card driver are the data rate used and whether the network card shall be active at all. In case of Linux, these parameters are commands sent by the operating system to the driver, which in turn configures the network card appropriately.

While real systems set these parameters while they are running, the concept in ns-3 is to set these parameters statically during simulation setup. As a consequence ns-3 lacks interfaces which allow the dynamic changing of these settings during operation. The current status of this implementation is therefore that these parameters can only be set initially through ns-3, which also has advantages as the configuration happens at a central place. The network card driver and message exchange already implement the configuration of above parameters, so that only changes inside ns-3 are needed in order to provide this functionality.

Scanning

A similar problem arises with the process of scanning for available wireless networks. Operating systems provide some mechanism to probe for available networks and then let the user or an automated tool decide to which to connect. ns-3, however, supports scanning for available networks only in that way that it probes for the statically configured SSID and immediately connects to the first it finds. Therefore, the scanning functionality is missing in this implementation. As with the configuration of parameters, the Linux driver and message exchange are prepared, so that only ns-3 would have to be adapted.

Statistics

In reverse to the configuration of the parameters of a wireless network card, it is also possible to query the network card for its current status. In addition to the settings which can be configured through the network driver, the virtual network card also reports the signal quality of the associated network. In case of infrastructure networks, this is the signal strength of the access point connected to. In case of ad hoc mode networks, the so-called *spy interface* of Linux can be used to request the signal quality on a per station basis.

4.2 Message Exchange

In order to connect these different systems with each other, a common format for the data exchange is required. Below follows a brief description of the messaging mechanism which is used to transfer the required information.

Since the network simulation is a requirement for the overall setup, it has to be run first and serves as a central coordination point. When a driver is loaded, it registers at the simulation so that a connection between them is established. From then on, messages can be exchanged in both directions whenever necessary. From this point of view, the simulation acts as a server and the drivers as clients. In order to identify the different driver instances, each of them is assigned a client identifier beforehand. Using this number enclosed in each message sent and received, the network simulation can match messages to the corresponding gateway nodes.

The second type of information enclosed with every message is a type identifier which is used to distinguish between the different messages that are used for different purposes. The message type and the client identifier are located at the beginning of each message, so that they can always be processed, regardless of the content of the rest of the message. This has the advantage that different modules inside the simulation or drivers can process this first part of a message and then hand it over to a more specialized function.

Registration process

Three different messages are used for the registration process: When the driver is loaded, it sends a registration request to the simulation. Using the client identifier, the simulation looks up the corresponding gateway node and responds with initial data required to set up the driver. When the driver is unloaded later-on, it sends an unregister message to the simulation, so that it knows that the driver cannot be reached anymore.

Exchange of data frames

Data frames have to be exchanged whenever a frame shall be sent or is received by the wireless network device. These messages simply consist of the general header and encapsulate an Ethernet frame, thus containing all data required for further processing. An exception is the monitor mode, where all frames received by the wireless network card are sent to the driver, including the 802.11 MAC header and a radiotap header [86] which provides additional information like the channel, timestamp and signal strength which with the frame was received.

In order to support the virtual buffer described above, the network simulation notifies the driver whenever the network card sends a frame which was scheduled beforehand. This message contains just the size of the frame, but is enough for the driver to calculate how many bytes are still enqueued for transmission. As this feature is optional, the corresponding program code is only executed if the driver states this in the initial registration message.

Statistics updates

The wireless driver can be queried for statistics of the wireless network card (its current settings and the received signal strength), which are only available in the simulated network card. Thus, this information has to be transferred from the simulation to the driver at some point. One way to implement this would be to have the driver request this information every time it is needed, which would cause a small delay until the information is available in the driver. However, the wireless network interface used by Linux splits this required information into several commands, so that a number of message exchanges would be required, whose delays would sum up. The option which this implementation therefore uses is to directly push all available statistics data to the driver. This means that initially and whenever new data is available, a statistics message is sent to the network driver, which then has all data available in order to respond to queries by the operating system.

Spy mode updates

The spy functionality, which delivers the signal strength of ad hoc networks on a per-station basis, is initially disabled. In order to use this feature, the driver sends a message to the simulation in which the corresponding MAC addresses are listed. From then on, the network simulation sends update messages which contain the MAC address and its corresponding signal quality.

Network card configuration

The configuration of the simulated network card happens through several different messages. When the driver is requested to set a specific parameter of the network card (channel, mode, BSSID, SSID, data rate or status), it issues a corresponding request message to the network simulation. After configuring the corresponding option, the network simulation replies with a response message indicating whether the operation was successful or not. Besides giving feedback whether the requested configuration could take place, the response message is used to block the corresponding request of the operating system until the operation has been performed.

Scan requests

Like the configuration of parameters, scanning only takes place when it is requested by the operating system. The driver sends a message to the network simulation in which it is stated whether the scan shall be active or passive[1] and optionally the SSID to scan for. After the network simulation has acquired a list of available networks, it is sent in a response message to the network driver which hands it over to the operating system.

4.2.1 Transport Mechanisms

The message format described above defines how the network simulation and driver communicate with each other. In order to so, it is required to utilize some transport mechanism which encapsulates these messages and exchanges them between both systems. Although in principle any technique for data exchange can be used, it is not an arbitrary decision, because the utilized transport mechanism influences the behavior of the overall system.

An important point is that the abovementioned message format does not include any acknowledgments or protection protection mechanisms. One requirement is therefore that the message exchange between simulation and the driver happens in a reliable fashion. Furthermore, the delay introduced through the transport mechanism should be as small as possible, because it can be perceived as an additional delay in the emulated network.

A different aspect is how flexible the simulation and the driver can be connected with each other. As discussed in Section 3.3.3, network connections are very flexible in this regard and allow the communication on one machine and across distributed systems.

4.2.2 UDP Transport

The transport mechanism which is used as part of this implementation is the user datagram protocol (UDP) [82]. The advantage of UDP is that it is very lightweight and allows to send messages directly without the need of a connection establishment. Furthermore, UDP is widely supported and can be accessed both through the Linux driver and the ns-3 network simulation. As thus, it is an easy to implement, yet very flexible solution.

There are two potential drawbacks of using UDP as a communication method. The first one is an additional delay which might be introduced through the processing of

[1]Passive scanning means that the network card does note issue a probe request, but just listens for beacon messages announcing wireless networks.

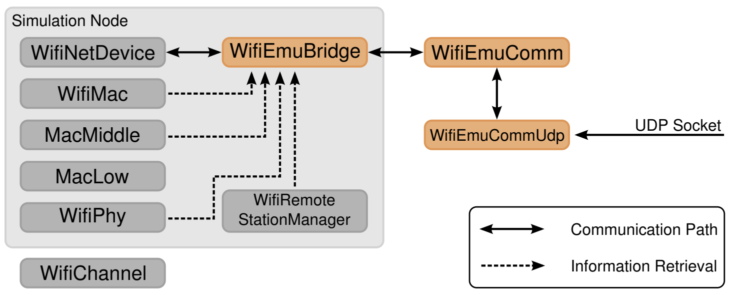


Figure 4.2 Three additional classes implement the required functionality in ns-3.

network packets in the network stacks used by the simulation and driver to communicate with each other. The other drawback is that UDP does not give any guarantees that a message indeed reaches the recipient. While other transport protocols such as TCP [15] can give such guarantees, this does happen through retransmissions which introduce additional delays. Thus, the simulation and drivers should in any case not be connected through an unreliable network, as large delays can be perceived by the system under test and diminish the realism of the emulation system.

4.3 Extending ns-3

As mentioned above, the network simulation tool ns-3 [103] is used to implement those parts of the system that reside in the network simulation. ns-3 was chosen because of its good support for data exchange with real systems. It already contains a real-time scheduler, which is a requirement to exchange network packets with external systems. Furthermore, the internal representation of network packets is the same as in real networks, so that no explicit conversion is required when packets are exchanged with real systems.

The 802.11 network model used in ns-3 is based on the model used in a network simulation tool called yans [64]. It consists of a detailed MAC implementation and a PHY layer for 802.11a and 802.11b networks [78]. It furthermore supports optional QoS (802.11e) and infrastructure as well as ad hoc networks. The ad hoc network implementation, however, is not complete, in that it currently only sends the data frames themselves and does not contain management operations. Still, it can be used for the simulation of ad hoc networks, even though the behavior is not compliant with real 802.11 networks. The complete Wi-Fi model consists of several classes which form a stack of sublayers.

As intended by abovementioned concept, emulation bridges are used for the functional integration of gateway nodes inside the simulation. The concrete instantiation of this concept has been added to ns-3 in the form of three new classes: `WifiEmuBridge` contains all functionality which has to be performed for each single gateway node used in the simulation. The general abstract functionality required for communication with external systems is managed by `WifiEmuComm`, whereas `WifiEmuCommUdp` refines this functionality for the use of UDP as transport mechanism. The way in which these classes interact with each other and integrate into the wireless network stack is shown in Figure 4.2.

In principle, each gateway node could establish the connection with the corresponding network driver on its own and all functionality could be implemented in a single class. However, all gateway nodes would then make use of a dedicated interface (e.g. a device file [19] or a network socket [109]), so that separate resources of the operating system hosting the network simulation would be required for each gateway node. By outsourcing this communication-dependent functionality into a separate class which acts as proxy for the communication of all gateway nodes, only a single interface is required for the complete simulation. This communication-specific functionality is implemented through the class `WifiEmuComm`. This class itself is abstract and only contains functionality which is independent of the transport mechanism used to exchange messages. Derived subclasses, such as `WifiEmuCommUdp`, add the functionality required for specific transports and make the whole class functional. Through this mechanism, only parts of the functionality have to be implemented when additional transport mechanisms are added.

The rest of this section describes the functionality provided by each of the three classes. A more practical guide on how to use the system, can be found in Appendix A.1.

WifiEmuComm

The purpose of the class `WifiEmuComm` is to organize the message exchange between the single `WifiEmuBridge` objects and the corresponding real systems in such a way that only a single communication interface is utilized. This is achieved through the following mechanism: All instances of `WifiEmuBridge` register at `WifiEmuComm`, which creates exactly one instance of `WifiEmuComm` (or more precisely, a subclass of `WifiEmuComm`). By utilizing the functionality provided by the corresponding subclass, this single instance from then on handles all communication with external systems. Furthermore, all registered `WifiEmuBridge` objects are added to a map containing their client identifier. This map is later-on used to match messages received from real systems and to find corresponding communication endpoints for the sending of messages.

While the sending of messages is performed entirely by the subclass, parts of the receiving functionality is performed by `WifiEmuComm`. A common characteristic of interfaces used for the communication with external systems is that the reception of data is performed using a function which blocks until new data is available. As calling such a function directly within ns-3 would block the complete simulation, an extra computation thread is started by `WifiEmuComm`. This thread waits until new data has arrived, looks up the corresponding `WifiEmuBridge` object and forwards the message. This procedure runs in an infinite loop as long as there are `WifiEmuBridge` objects registered.

The fact that a separate thread is used to receive data is another argument in favor of the use of a single communication interface: If each gateway node handled the communication on its own, a separate thread would be required for each gateway node. In consequence, the approach of using only a single communication interface therefore saves operating system resources both in form of the communication interface itself and computation threads.

WifiEmuCommUdp

As parts of the required communication functionality are already handled by the abstract class `WifiEmuComm`, its subclasses only have to implement the functionality which depends on the transport mechanism being used. This comprises the initialization of the communication interface used and the sending and receiving of data. In order to fulfill this functionality, `WifiEmuCommUdp` makes use of the Berkeley Socket [50] interface.

WifiEmuBridge

The class `WifiEmuBridge` fulfills the main functionality of the ns-3 implementation: It acts as a bridge between a `WifiNetDevice` and a real implementation, whereby it makes use of the functionality provided by `WifiEmuComm`. By being installed on a network device and thereby replacing the upper layers of a node, its overall concept is similar to the TapBridge (see Section 2.5.2) which ns-3 provides. The details, however, differ widely because `WifiEmuBridge` has to fulfill a number of additional tasks which are related to wireless networks. Furthermore, the interfaces used to perform the integration of external systems are very different.

During simulation setup an instance of `WifiEmuBridge` is installed on each of the gateway nodes, which means that various callbacks are set in the stack of the wireless network card. Moreover, the object registers at `WifiEmuComm` and is thereby ready to receive messages. The `WifiEmuBridge` is then idle until it receives a registration message of the corresponding network driver to which it replies with initial statistics of the `WifiNetDevice`, which the driver needs for initialization. From this point on, the emulation for the corresponding node is set-up and running.

As the main task of `WifiEmuBridge` is to act as a bridge between the `WifiNetDevice` and the corresponding driver, data frames have to be forwarded in both directions. Data frames received through `WifiEmuComm` are simply handed over to the corresponding sending function of the `WifiNetDevice`. In the other direction, a callback forwards all frames received by the `WifiNetDevice`, which then encapsulates them in the corresponding message and sends them to the driver.

While the pure sending and receiving of data frames is directly performed through the `WifiNetDevice`, a number of other callbacks and getter-methods is used to extract various information from other classes of the wireless network card's protocol stack. For some information, existing callbacks can be used, but new callbacks were added as well in order to extract the required data. Besides these simple changes, the Wi-Fi model of ns-3 did not have to be modified.

The information obtained through these callbacks is mainly used for the statistics sent to the driver. Whenever such a callback is called, the statistics are updated accordingly and sent to the driver. The most important sources are the raw 802.11 frames which `WifiPhy` forwards alongside with their signal strength. In monitor mode, a radiotap header is added to them and they are forwarded to the network driver. In infrastructure mode, the signal strength of beacon messages sent by the access point is used to calculate the signal quality, which is then reported to the driver. Likewise, the signal strength of received messages is used for the spy functionality in ad hoc mode. Information acquired through the other classes includes

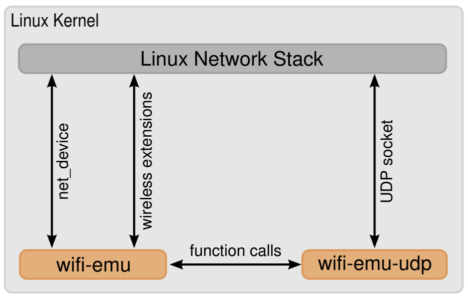


Figure 4.3 The Linux driver emulating a wireless network card consists of two kernel modules.

the information whether a connection is currently established, the mode of operation, the channel number, the PHY standard, BSSID, SSID and the current data rate. If the network driver wants to make use of the virtual buffer feature, another callback is used to report frames which are actually sent over the channel.

As described above, the configuration of the network card currently happens through the regular ns-3 mechanisms during simulation setup. The corresponding methods which receive the requests sent by the driver simply reply that the operation failed. In order to support the configuration through the driver, the Wi-Fi stack of ns-3 would have to be modified in a more invasive way. Similarly, requests for a scan of available network are replied with an empty list of scan results, because the required logic is missing in ns-3's Wi-Fi model.

4.4 Linux Driver

The network driver which is part of this implementation has been written for the Linux [102] operating system. The fact that Linux is open-source makes it ideal for research, as all parts of the system can be inspected and modified. Thus, the use of Linux eases the development and evaluation of this system itself. In addition, the developed driver can be used in the many research projects which make use of Linux.

Since the main goal of this driver is to represent the simulated wireless network card, it has to interact with the Linux network stack in the same way a driver of a regular wireless network card does. While a regular network card driver interfaces on the other side with the network card hardware, this driver uses UDP network communication to exchange messages with the network simulation.

As Figure 4.3 shows, the functionality has been split into two kernel modules. Similar to the split inside the network simulation, the functionality dependent on the specific message transport used to communicate with the simulation has been outsourced to a separate module. The kernel module `wifi-emu-udp` handles the communication with the network simulation, whereas `wifi-emu` implements the interfaces required to act as a wireless network card.

A system used for the wireless network emulation loads the two kernel modules during start-up or at some later point in time. Using the message exchange mechanism provided by `wifi-emu-udp`, `wifi-emu` registers at the network simulation and afterwards registers itself as wireless network driver in the Linux kernel. From then on, the kernel modules provide the impression of a wireless network card and the system can use the simulated wireless network.

The remainder of this section describes the functionality of both kernel modules in more detail. A description of how to use and configure the kernel modules in a wireless network emulation scenario is given in Appendix A.1.

wifi-emu

The main purpose of the module `wifi-emu` is to act as a regular wireless network card driver. As such, it has to make use of the interfaces Linux provides to access wireless network cards, whereby 802.11 wireless network cards are handled by Linux through the same kind of interface as Ethernet network cards. This interface, which is depicted as *net_device* in Figure 4.3 and described in [19, 111], works as follows: During initialization or when hardware is found, a network card driver registers itself at the networking subsystem, providing a list of function pointers. These functions are later-on called by the networking subsystem to pass data which has to be sent, to retrieve statistics or to start and stop the network card. In turn, the network driver can call functions of the networking subsystem to start and stop its sending queue or to transfer received packets.

While this general network card interface already allows the driver to exchange network packets with the networking subsystem, it does not support any wireless network card specific features. For this purpose, the so-called *wireless extensions* [115] (wext) are added on top of this interface. Through a number of additional pointers to functions provided by the network driver, the Linux kernel can set or get additional parameters or retrieve statistics of the wireless network card. From Linux version 2.6.22 on [67], a new interface called *cfg80211* [16] can be used instead of the wireless extensions. This implementation makes use of the old wireless extensions, as they provide all necessary features and are available in old as well as new Linux kernel versions.

Other kernel interfaces used by this driver, however, have changed in the course of different kernel versions. In order to make this driver compatible with different kernel versions, old and new versions of the required interfaces are used in the program code and selected through conditional compilation. Hereby, the driver can be compiled and used for different kernel versions.

As the module `wifi-emu` does not stand on its own, but makes use of the communication functionality provided by `wifi-emu-udp`, it also has to interface with this module. This happens through function calls between the two modules. Initially `wifi-emu-udp` calls an initialization function of `wifi-emu`, so that it can start the initialization, including the registration at the network simulation. Later on functions are used to send data through `wifi-emu-udp` to the network simulation and to pass received data to `wifi-emu`.

Simply put, the task of of `wifi-emu` is to connect between these different interfaces and to translate the data exchanged between them. By forwarding data frames received from either direction, the wireless network connection itself is established.

Frames received by the simulation are simply passed to the Linux network stack, which perceives it as a frame received by the wireless network card. In the opposite direction, the Linux networking subsystem passes frames which shall be sent by the wireless network card[2]. These are forwarded to the simulation, which then sends them over the wireless channel. If the virtual buffer feature is enabled, the driver counts the number of bytes sent to the network simulation and later-on receives a notification that this number of bytes has been transmitted. If this counter exceeds a configurable threshold, the virtual buffer is full and the driver instructs the Linux network stack to block the transmission of new frames.

Through the wireless extensions, the Linux network stack also passes various other requests to the network driver. In the case of statistics to provide for the wireless network card driver, `wifi-emu` can immediately respond with the statistical data pushed from the network simulation. For calls regarding the configuration of the network card or the scanning of networks, `wifi-emu` issues a corresponding request to the network simulation and blocks the function call until the corresponding reply message is received.

As mentioned in Section 2.3.2, most wireless network cards allow only the sending of frames with up to 1500 bytes, although the standard allows for 2296 byte. In order to resemble this fact, the *maximum transmission unit* (MTU) of the network card driver is initialized to 1500 bytes, but can later-on be changed to sizes up to 2296 bytes.

An interesting point is that the module `wifi-emu` itself is completely passive and all these actions happen only through functions being called by the network stack or `wifi-emu-udp`.

wifi-emu-udp

The module `wifi-emu-udp` is very simple and fulfills only one single task, namely to establish an UDP network connection to the simulation. In contrast to `wifi-emu` it is active on its own through a number of mechanisms. When the module is loaded, it first initializes itself and then calls the initialization function of `wifi-emu`, which initializes the network card and registers itself at the simulation.

In order to receive packets from the network simulation, a separate computation thread is used. This thread runs in an infinite loop waiting for UDP packets to arrive, whose content is then forwarded to the module `wifi-emu`. For the other direction, `wifi-emu` calls a sending function with messages which shall be sent to the network simulation. However, an internal policy of the Linux kernel forbids accessing network sockets from within calls made by the network stack. As those functions of `wifi-emu` which initiate a transmission to the simulation are called from within the network stack, it is therefore not possible to send the corresponding message directly. Therefore, messages to be sent are first put into a queue, and then sent by another computation thread used for that purpose. In summary, the consequence of these mechanisms is that the module `wifi-emu-udp` starts two separate computation threads for the data exchange with the network simulation.

[2]As 802.11 networks are compatible to Ethernet, Linux uses Ethernet frames to exchange data frames with wireless network drivers

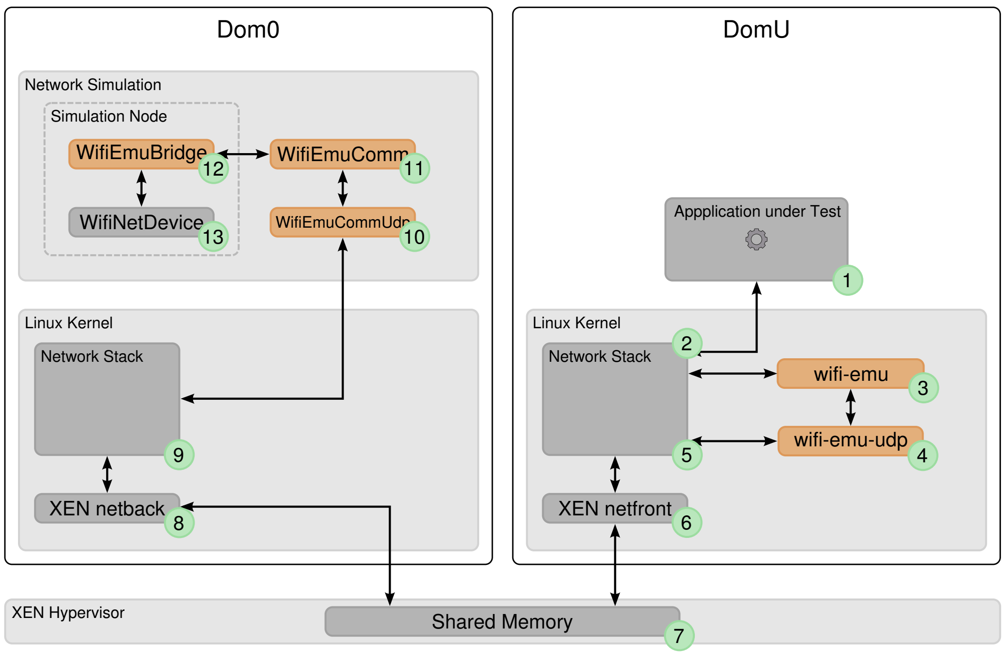


Figure 4.4 A packet traverses the wireless network emulation system using Xen paravirtualized machines.

4.5 Setup Example

The focus of this section is to show exemplarily how a complete system including all components can be assembled. Figure 4.4 depicts a setup using the Xen Hypervisor [118] to run all components on a single machine. The ns-3 network simulation is run as part of a Linux system inside Dom0. An additional paravirtualized DomU system is used to run a Linux system belonging to a gateway node inside the network simulation. As part of this Linux system, the `wifi-emu` driver is used to provide a virtual 802.11 network card.

Assuming the application under test sends a packet using the wireless network card's interface, it takes a number of steps which are depicted in Figure 4.4. While most of these single steps have already been described above, two facts are worth mentioning: The first one is that the packet passes through the Linux network stack of the DomU system twice. First, the application under test hands a network packet to the network stack (2), which after a number of modifications hands an Ethernet frame to the wireless network card driver (3). The driver then encapsulates this frame in its message format, which it hands for a second time to the network stack (5), this time as a UDP message destined for the network simulation.

The second noteworthy fact is how this UDP message is transferred to the system running the network simulation. After the network stack has added the required headers, it hands an Ethernet frame to the network card driver provided by Xen (6). This driver has a corresponding counterpart providing a network interface in the Dom0 system (8). In order to transfer the data between these two network interfaces, Xen makes use of a shared memory mechanism provided by the hypervisor (7). As a consequence, the data transfer between the wireless network card driver indirectly uses shared memory to exchange data with the network simulation, as it has been proposed as a design alternative in Section 3.3.

One aspect not shown in Figure 4.4 is how the different time concepts of the real system and the network simulation are brought together. The easiest solution which can be used without any additional modifications is to use the real-time scheduler provided by ns-3. This approach, however, can only be used as long as the system is fast enough to process the network simulation in real-time. As an alternative, synchronized network emulation [112] can be used to synchronize the execution of network simulation and the DomU systems.

5 Evaluation

The aim of this chapter is to evaluate the concept and implementation which have been developed as part of this thesis. In order to do so, a number of tests using the exemplary implementation have been performed.

The overall goal of this concept, namely to ease the development of software which makes use of wireless networks, is hard to quantify. The same applies to the design goals which were defined in Section 3.1, for which reason it is not possible to evaluate them one by one. The following tests therefore cover multiple aspects at once and a summary at the end of the chapter discusses the results in a more general fashion.

All tests were performed on two Dell Optiplex 960 machines, each equipped with a 3 GHz Quad Core CPU, 8 GB of main memory, a 320 GB hard disk and two gigabit network cards. One of the network interfaces was used to connect the two machines directly, the other one for remote access. Ubuntu 8.04 Server Edition was used as operating system in conjunction with two different Linux kernels, a natively executed Ubuntu 2.6.24 kernel and a 2.6.18 kernel with Xen support (used for Domain 0, as well as paravirtualized machines). The Xen system was used whenever virtual machines were involved in the tests and was patched to support synchronized network emulation [112]. Inside the virtual machines, a Debian Lenny base system was used. For the simulation part, ns-3 was used in its version ns-3.7 and was compiled with the optimized build profile.

The remainder of this chapter is structured as follows: The results of some basic measurements are presented in Section 5.1. After that, Section 5.2 discusses a larger emulation scenario, resembling measurements taken in a real outdoor ad hoc network. To conclude, Section 5.3 discusses the obtained results and experiences from a broader perspective.

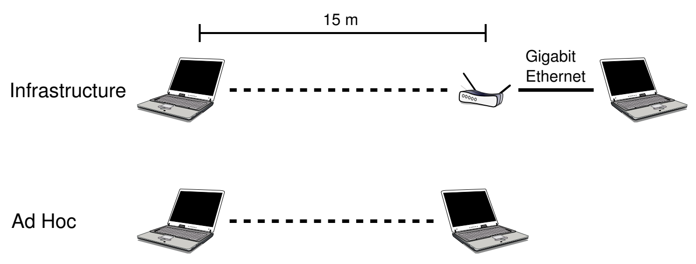


Figure 5.1 The test setup for the basic 802.11 network measurements.

5.1 Basic Measurements

As this thesis proposes wireless emulation as an alternative to measurements using real testbeds, a key question is whether the behavior of an emulated wireless network is equal to real wireless networks. Two often used metrics which characterize any network link are its maximum throughput and time delay. Since they determine the limits of a network connection, these two properties have a large influence on the behavior of many network protocols.

For the performed throughput measurements, *netperf* [104] version 2.4.4 was used with its `TCP_STREAM` and `UDP_STREAM` tests, measuring the achievable throughput between a client and a server with the TCP and UDP transport protocols. In order to increase the reliability of the measurements, the confidence interval calculation provided by netperf was used. In this mode, netperf repeats the test until the mean of the measured throughput is within a certain confidence-interval width. Netperf was configured to reach a 5%-confidence interval width with 99% confidence. Thus, the resulting mean value of the throughput derives with 99% confidence not more than ± 2.5% from the real mean value.

For delay measurements, the *round trip time* (RTT) between two systems was measured using the Linux *ping* tool. It sends an ICMP Echo Request to the remote host, waits for the answer and measures the time between request and response. As the link used to connect the two systems is traversed twice, the RTT is approximately twice the link delay. The ping utility, which was modified to output the measured RTT with microsecond accuracy, was configured to take 1000 measurements with an interval of 0.2 s.

5.1.1 Comparison to Real Measurements

As a first test, the throughput and round trip time of an emulated wireless network were compared to those of a real wireless network. As the implemented system is capable of emulating 802.11a and 802.11b networks in infrastructure and a (limited) ad hoc mode, the measurements were performed for these four types of networks. The setup used for these measurements is depicted in Figure 5.1: Two laptops, respectively a laptop and an access point, were located 15 m apart from each other. In the infrastructure scenario, the second laptop was connected via Gigabit Ethernet to the access point and served as additional communication endpoint for the measurements.

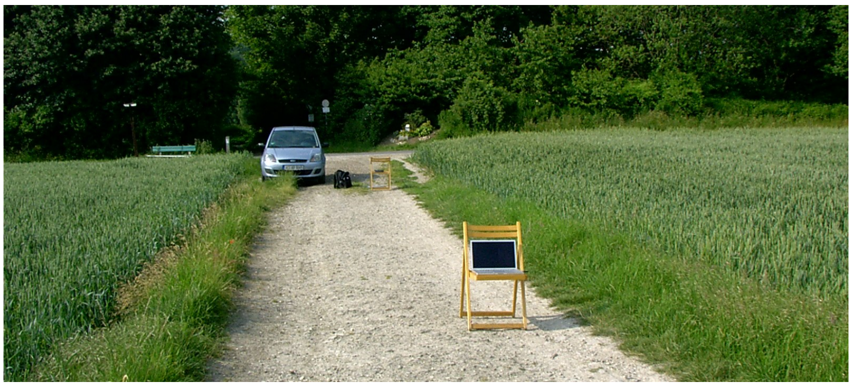


Figure 5.2 Picture taken during the real measurement.

Setup

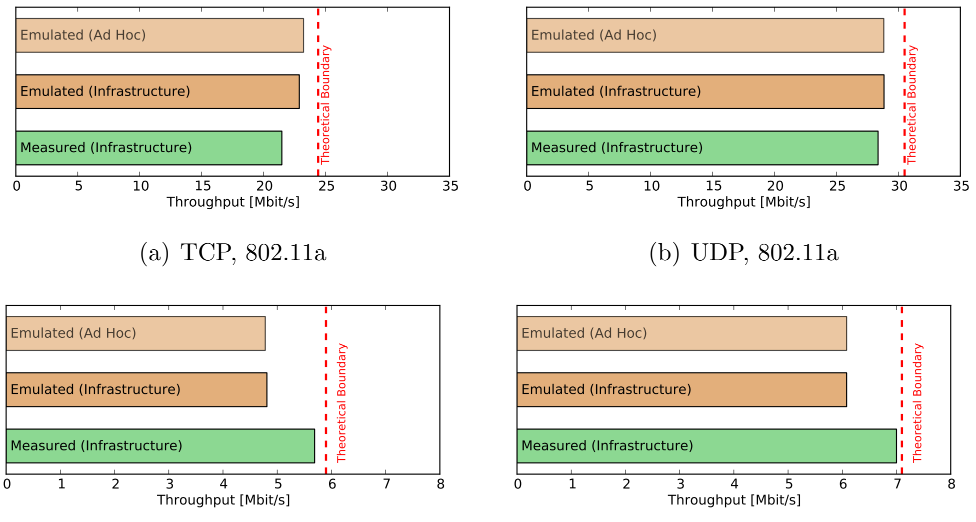
In order to obtain reference values, the throughput and RTT of real 802.11 wireless networks were measured. These measurements were performed on a field outside the city, so that the disturbing influence of interference was minimized. Figure 5.2 shows a picture of the measurement setup.

For the real measurements, two MacBook Pro laptop computers with Ubuntu 10.04 Desktop were used. Each of them was equipped with a dual-band Wi-Fi card (2.4 GHz and 5 GHz). Unfortunately, the Wi-Fi cards did not allow to set up a stable ad hoc network connection. Therefore, only measurements for the infrastructure scenario were performed. To that end, one of the laptops was wirelessly connected to a Linksys WRT610N wireless router, which was configured to provide an 802.11b network on the 2.4 GHz band and an 802.11a network on the 5 GHz band. Furthermore, all Layer-3 functionality, like DHCP or routing, was disabled in the router, so that it only served as access point. The other laptop was connected via Ethernet to the router and had all wireless functionality disabled.

The laptop with the wired connection ran the netserver component of netperf and the other one was used to execute the tests using netperf and the modified ping utility. The throughput was measured between both laptops, whereas the round trip time was measured between one laptop, the access point and the other laptop. All encryption mechanisms of the wireless devices were turned off during the tests, because the emulation environment does not support any encryption.

The fact that no reference measurements for 802.11 ad hoc networks could be performed does not strongly influence the following evaluation: 802.11 networks in ad hoc and infrastructure mode only differ with regard to management operations, but perform the actual data transmission in the same way. Furthermore, ns-3 does not provide an authentic ad hoc mode implementation, so that the emulation results for ad hoc mode have to be treated with care anyway.

Using the same setup, the measurements were repeated in an emulated 802.11 wireless network. Instead of the laptop computers, XEN paravirtualized machines (PVMs) were used. The emulated wireless network card driver provided them with a wireless network interface. For the case of infrastructure networks, one virtual machine used the Ethernet device which was directly provided by the virtual machine.



(a) TCP, 802.11a (b) UDP, 802.11a

(c) TCP, 802.11b (d) UDP, 802.11b

Figure 5.3 Throughput of 802.11 networks measured in a real experiment and in an emulated network. (theoretical boundary values taken from [6])

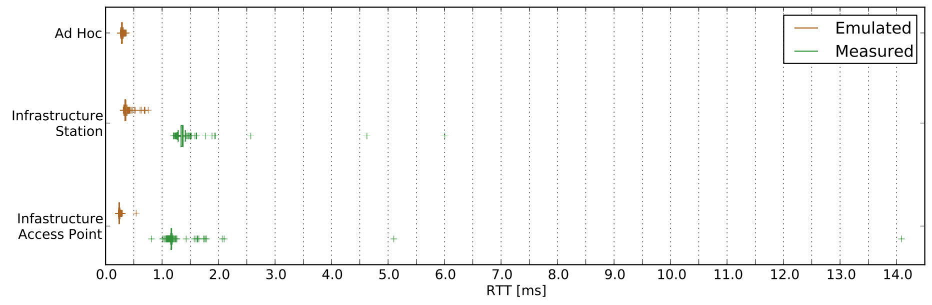
Both virtual machines were run together with the network simulation on a single test machine.

For the ad hoc case, the network simulation included only two wireless stations on which a `WifiEmuBridge` was installed. In the case of infrastructure networks, the simulation consisted of three nodes: One wireless station with a `WifiEmuBridge`, one simulated access point and a third node on which a `TapBridge` was used to connect to one of the virtual machines. The access point and the third node were connected by a duplex CSMA channel, thereby simulating the Ethernet connection of the real setup. In both cases, the network simulation modeled the wireless link with a two ray ground wave propagation model [87]. The network simulation was run using ns-3's real-time scheduler, configured to abort the simulation in case it is too slow to keep up with wall clock time.

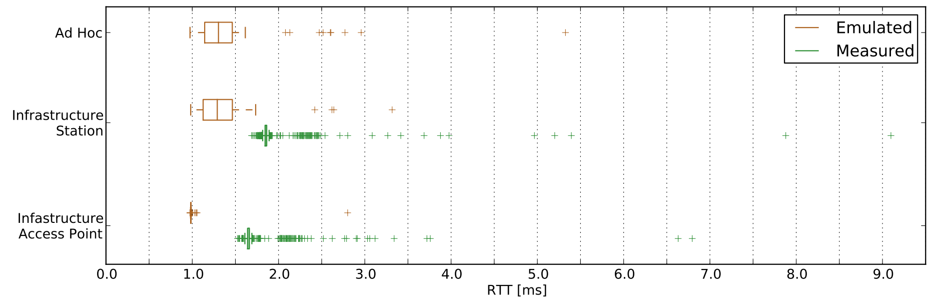
Results

Figure 5.3 shows the measured throughput of 802.11 networks. The theoretical boundary values have been taken from [6] and denote the maximum possible throughput using the respective networks and transport protocols. The measured throughput is in all four cases below the theoretical boundary, which shows that the results are not altogether wrong. Furthermore, the throughput of emulated ad hoc networks is almost identical to the one of emulated infrastructure networks. This shows that the simple ad hoc implementation of ns-3 does not influence the network with regard to throughput.

For 802.11a, the emulated networks are slightly faster than the real measurements. For 802.11b, it is the other way round and the real measurements came much closer to the theoretical boundary. Measurements presented in [6] show that the achieved throughput in 802.11 networks is highly dependent on the used network hardware.



(a) 802.11a



(b) 802.11b

Figure 5.4 Round trip times of 802.11 measured in a real experiment and in an emulated network. (whisker length: 1.5 IQR)

Therefore, the deviations between emulated networks and real measurements are acceptable.

The corresponding measurements for the round trip time are depicted as boxplots in Figure 5.4. Most apparent in this figure are the outliers which occurred throughout all measurements. The real measurements exhibit far more outliers, which is probably caused by the wireless channel. Although the measurements were performed on a site with low interference, there certainly was some disturbance which has not been modeled through the simple wireless channel of the simulation.

More interesting than the outliers are the differences of the median value. One apparent effect occurs in the measurements of the infrastructure scenario: The round trip times of ICMP echo requests sent to the other station are always a bit higher than those sent to the access point. As an additional Ethernet connection is used in the first case, it introduces an additional delay which can be seen in the results.

The round trip times measured in emulated networks are in all cases significantly lower than the corresponding values which were measured in the real experiment. It is likely that the reason for this difference lies in the oversimplification of the wireless simulation model (a more detailed analysis on the composition of measured round trip times follows in section 5.1.3). The relation of the different values to each other, however, is correct. This leads to the conclusion that the emulated network systematically underestimates the communication delay, but is apart from that correct.

During all of the aforementioned tests, the virtual buffer of the emulated wireless network cards was enabled. Its function can be easily observed in throughput measurements using UDP. If the virtual buffer is disabled and the netperf measurement is started, the network simulation, to which the driver is connected, aborts stating that its internal simulation time derives too much from the wall clock time. What happens is that the driver accepts outgoing network packets at a very high speed and forwards them to the network simulation. The network simulation enqueues these incoming messages as events, but cannot process them fast enough too keep up with the rate of incoming events. As the real-time scheduler of ns-3 is configured to abort the simulation when the time drift between simulated and wall clock time becomes too large, the simulation exits. When using the `TCP_STREAM` test of netperf, these effects do not occur as the congestion avoidance algorithm of TCP detects this situation.

5.1.2 Influence of Virtual Machines

The results of the last section show that the measurements which were performed using the emulation environment approximately match the real measurements. As virtual machines have been used in the emulation setup described above, a key question is whether the use of virtual machines has an influence on the measurements. In order to evaluate this question, the emulation measurements of abovementioned infrastructure scenario were repeated[1] using different execution mechanisms: Native execution (i.e. without making use of virtual machines), paravirtualized machines (PVMs) and hardware-virtualized machines (HVMs). The experiments for both types of virtual machines were conducted using real-time execution and by making use of synchronized execution, thus resulting in a total of five different execution types.

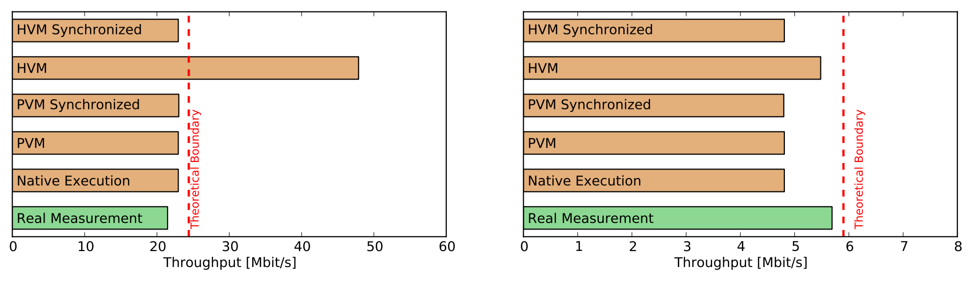
Setup

For the experiment using native execution both test machines were utilized. One machine ran just the wireless network card driver, while the other machine executed the network simulation. The two network interfaces used to perform the measurements were thus the wireless network card on one machine and the tap-interface created by the network simulation on the other machine. To utilize just one computer for this setup was not possible, because the Linux networking subsystem does not support communication between local network interfaces[2].

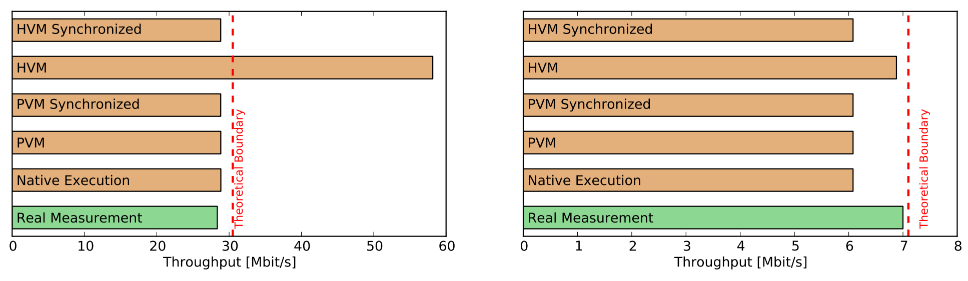
The experiment using PVMs and real-time execution was the same as in Section 5.1.1. In the case of HVMs, only the type of virtual machine was changed. For the remaining two cases synchronized network emulation, as presented in [112], was used. In short this technique works as follows: The virtual machines and the network simulation connect to a central *synchronizer* which allows them to run for a specified amount of time. When they are finished with this *time-slice*, they pause

[1]This time, the round trip was only measured between the two stations and not between one station and the access point.

[2]As reported in [84] this behavior can be changed through patching the Linux kernel, but for simplicity two machines have been used for this setup



(a) TCP, 802.11a (b) TCP, 802.11b



(c) UDP, 802.11a (d) UDP, 802.11b

Figure 5.5 Throughput of emulated 802.11 networks, measured using native execution and different types of virtual machines.

execution and report this to the synchronizer, which waits for all components to finish and then assigns the next time-slice. Simply put, one could say that the virtual machines are run in simulation time, which is contrary to the usual approach of running the simulation in real-time. The advantage of this approach is that it allows to run emulation scenarios which cannot be executed in real-time because the simulation is too slow. Although this is not the case for the simple experiments used here, it might be interesting to see how this execution method performs in comparison to real-time execution. For these experiments, the synchronizer was configured to assign time-slices of 100 µs.

Results

Figure 5.5 shows the throughput measurements of the five experiments and the real measurement. Leaving out the results of HVMs executed in real-time, the throughput of the remaining four execution types is almost identical, leading to the conclusion that the use of virtual machines does not influence the perceived throughput.

However, the throughput measured for HVMs not using synchronization deviates vastly. For 802.11b networks, the throughput is almost one Mbit/s higher, but still below the theoretical boundary. The values measured in 802.11a networks, however, are far off: They are not only above the theoretical boundary, but exceed in the case of UDP even the gross throughput of 802.11a, which is 54 Mbit/s. As such,

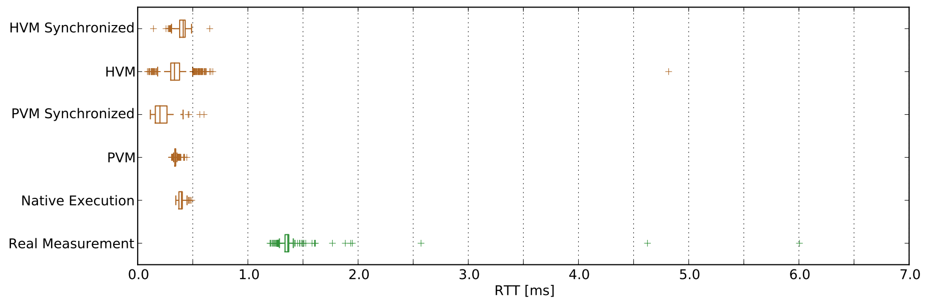


Figure 5.6 Round trip times of emulated 802.11a networks, measured using native execution and different types of virtual machines. (whisker length: 1.5 IQR)

the use of HVMs produced simply wrong results[3]. However, identical behavior could be reproduced by using a simple simulation consisting of just two nodes which were connected by a rate-limited CSMA channel. Thus, the wrong throughput of HVMs is not caused by the wireless network emulation itself. This issue clearly requires further investigation, but is outside the scope of this thesis as HVMs have only been used for this single experiment.

The round trip times of the 802.11a networks measured in these five experiments are shown in Figure 5.6. In contrast to the throughput measurements described above, the different types of virtual machines show a different behavior regarding the RTTs. Out of the five execution mechanisms, paravirtualized machines show the most stable behavior and have the smallest round trip times. When using native execution, the RTT increases slightly in value and variance. As an additional Ethernet connection has been utilized to connect simulation and driver, this increase is clearly allegeable to the delay caused by this connection. The round trip times measured for HVMs are slightly higher than the ones measured for PVMs, which is a general characteristic as similar measurements in [97] show.

When using synchronized network emulation in combination with PVMs, the variance increases. The reason for this is the way synchronized network emulation works: In each synchronization step, all synchronized components run independent of each other and are therefore decoupled in time. This time-drift is bound through the duration of a time-slice, which in this case was 100 µs. As a single ICMP message traverses multiple synchronization components, these effects can sum up. Another observation is that the median of the RTTs decreases when using synchronization. This effect is also caused by the synchronization. As network simulation and virtual machines progress in their own virtual time, only time spent inside the virtual machine or in simulation time contributes to the RTT. Time spent during calculations in the simulation and the host system can only be perceived if it causes a transferred message to be delayed across multiple synchronization steps. As a result, synchronization can both increase and decrease the perceived time it takes for a message to be transferred. In case of HVMs, the round trip time increased when using synchronization. As it is uncertain whether the results for the case not using

[3]In order to make sure that it is not netperf that is the source of error, the test was repeated using *Iperf* [52] as tool for throughput measurement. The results of Iperf were identical to those of netperf.

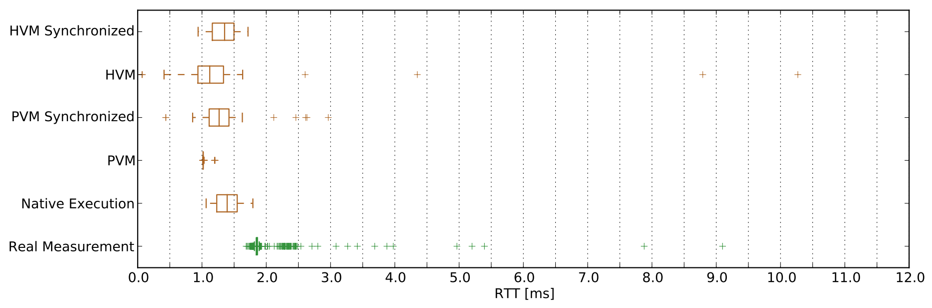


Figure 5.7 Round trip times of emulated 802.11b networks measured using native execution and different types of virtual machines. (whisker length: 1.5 IQR)

synchronization are correct, these results are not directly comparable to the other measurements.

Figure 5.7 shows the corresponding results measured for 802.11b networks. In comparison to the results for 802.11a, two differences are noticeable: For all cases except the paravirtualized machines not using synchronization, the variance of the measured RTTs increased. Another difference is that the synchronization of PVMs has led to increased RTTs, in contrast to the decreased values observed for 802.11a networks. A reason for the different behavior might be the scheduling of processes inside the single systems and across multiple (virtual) machines. This complex issue has not been further investigated and is of minor importance as the measured round trip times are still below the ones measured in the real experiment.

5.1.3 Detailed Time Analysis

A further experiment was conducted in order to investigate how the measured round trip times in emulated wireless networks compose. This question is of interest, because the communication delay between two nodes in regular network simulations is caused solely by the simulated communication channel. When using the proposed wireless network emulation environment, the simulated delay of the network simulation still exists, but additional delays emerge through the processing in real systems.

Setup

In order to inspect how much time is spent in which part of the system, the network simulation and network card driver were modified to log a timestamp whenever a network frame passes a certain part of the system. Using these modified components, 1000 ICMP echo requests were sent to the simulated access point of the abovementioned infrastructure simulation setup. In order to ease the collection of results, the network simulation and network card driver were run on the same system without making use of virtual machines.

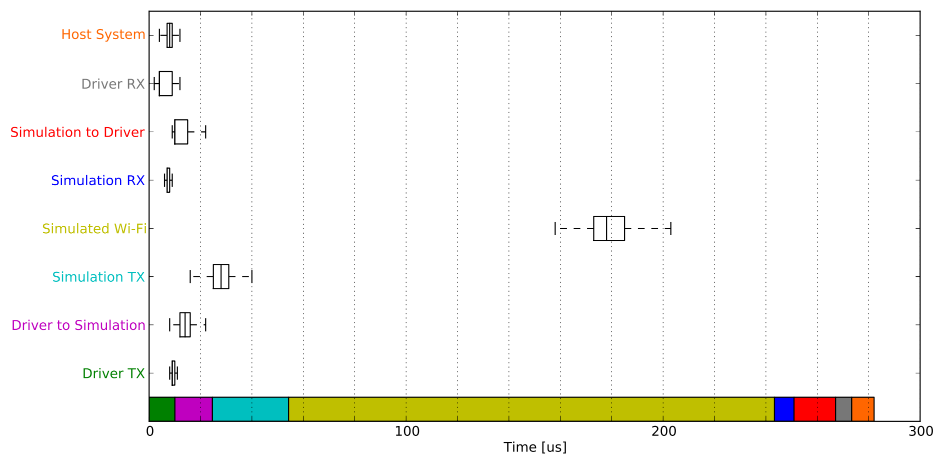


Figure 5.8 Composition of the round trip time in emulated 802.11a networks. (whisker length: 1.5 IQR, outliers not drawn)

Results

Figure 5.8 shows the results of these measurements for 802.11a: The single boxplots show how much time was spent in each part of the system and the bar at the bottom illustrates how these single times add up to the total measured round trip time. For better clarity, outliers have been left out in this plot. The corresponding version including outliers can be found in Appendix A.2.

The measurements start with the time marked by *Driver TX*, which is the time spent inside the driver for the processing of an outgoing frame. In this step the driver accepts a frame from the Linux network stack, encapsulates it in an UDP message destined for the simulation and hands this UDP message to the network stack. *Driver to Simulation* is the time it takes until this UDP message reaches the receiving thread inside the network simulation. The message is then processed in the wireless emulation bridge of the network simulation. This time is labeled with *Simulation TX*. After that, the frame is handed over to the simulated wireless network card and sent over the wireless channel to the simulated access point. The time it takes until the reply is handed over to the wireless emulation bridge is marked by *Simulated Wi-Fi*. The reply traverses the different components in the reverse order, which is labeled with *Simulation RX*, *Simulation to Driver* and *Driver RX*. The remaining time which is not covered by these measurements is labeled with *Host System* and is basically the time the network stack requires to process the ICMP Echo request and reply.

The clearly biggest part of the round trip time is spent in *Simulated Wi-Fi*, which contains the time required for channel access and data transmission. As this is the only time which would occur in a regular network simulation, the remaining time spent in other components can be seen as overhead caused by the emulation system. This time makes up a third of the complete round trip time of 802.11a networks, which is certainly quite high. The results of Section 5.1.1, however, have shown that the RTTs of emulated networks are far below the ones measured in a real network. The reason is that the network simulation only takes certain delays, like the channel access or signal propagation, into account. Other processing delays which are for example caused by the operating system of real systems are not reflected in a network simulation. Seen from this point of view, the time spent in the remaining components of the network emulation system is not a real overhead, but rather reflects time which would be spent in a real system as well.

Of further interest are the times *Driver to Simulation* and *Simulation to Driver*, which comprise the time it takes to transfer UDP messages between driver and simulation. As discussed in Section 3.3.3, an alternative to the message transfer using UDP would be the use of a shared memory mechanism, which would be more complicated to implement and restrict the use of the system to virtual machines running on a single physical machine. These results, however, indicate that such an approach would not be beneficial, as the local message transfer using UDP requires only about 30 µs.

The corresponding measurements for 802.11b networks are shown in Figure 5.9. They very much resemble the measurements for 802.11a networks, the only difference being that the time spent in the simulated wireless network is higher. This result is in line with expectations, as the simulated Wi-Fi link is the only change between the two setups.

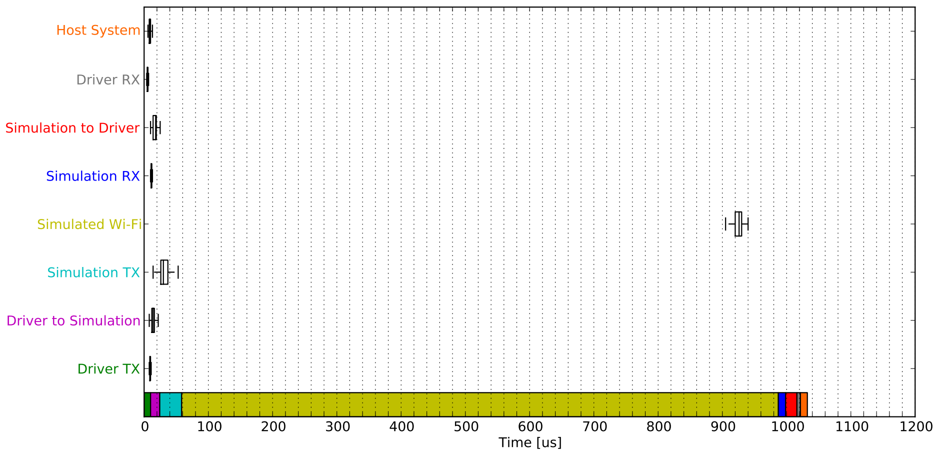


Figure 5.9 Composition of the round trip time in emulated 802.11b networks. (whisker length: 1.5 IQR, outliers not drawn)

5.2 Outdoor Scenario

The tests presented in Section 5.1 have shown that the throughput and round trip time measured in emulated networks approximately match those of real wireless networks. Although these two metrics are certainly important characteristics of a network connection, they do not describe all aspects of a wireless network. In addition, the experiments performed so far have been conducted in very small networks with only two wireless nodes. In order to supplement abovementioned measurements, a larger emulation setup using the ad hoc on-demand distance vector (AODV) [81] routing protocol has been evaluated.

In 2003, Gray et al. conducted an outdoor experiment [34] in which they compared the performance of four different ad hoc network routing protocols. The data which was acquired during this experiment was published later-on under the name *dartmouth/outdoor* in the CRAWDAD repository [20], which is a wireless network data repository. Furthermore, the authors repeated their experiment using a network simulation which made use of the same routing protocol program code. A further publication [68] describes the results of this network simulation in more detail.

As this work is well documented and the obtained data is publicly available, it was used as reference scenario for a wireless network emulation experiment. More specifically, the wireless network emulation environment was used to repeat the outdoor ad hoc network experiment. In combination with the simulation results published in [68], it is possible to compare the data which was obtained using wireless network emulation, to the results of the real experiment and the regular network simulation.

The outdoor experiment

Since the outdoor experiment of Gray et al. is used as reference for a similar emulation setup, some details about this experiment and the corresponding simulation are necessary.

The outdoor experiment consisted of 40 participants which each carried a laptop and moved randomly over an athletic field. During this experiment, four different ad hoc network routing protocols were run, each for a duration of 15 min. The laptops communicated using 802.11b wireless network cards which were set to a fixed datarate of 2 Mbit/s and made use of the so-called *ad hoc demo mode*. This mode is a predecessor of 802.11's ad hoc mode and differs from it in that it does not use any management frames.

A traffic generator was used to create UDP messages which were sent to randomly chosen destinations using the ad hoc routing protocol. Both the traffic generator and the software implementing the routing protocol created a log file containing information which amount of data was sent or forwarded at which point in time. Another file was created to log the physical position of each node, which was acquired from an attached GPS device. Additionally, all laptops broadcasted their current position to the network during the course of the experiment.

Seven out of the 40 laptops failed to generate data or routing traffic, so that effectively only 33 laptops took part in the experiment. As this was discovered after the experiment, the addresses of the seven failed nodes were still used as destinations for application traffic. An eighth laptop did not broadcast the position messages and was therefore excluded from the analysis.

In addition to the real experiment, the authors of [68] performed a network simulation resembling the outdoor scenario. They used their own network simulator called SWAN [98, 68] which executed the ad hoc network routing protocol implementations of the real experiment. This integration into the network simulation was achieved through source code modifications of the routing protocol implementations. This process, which the authors call *Direct Execution*, is described in [68].

The network simulation consisted of 33 nodes, resembling the ones which did not fail during the real experiment. It generated application traffic according to the log files created in the outdoor experiment, so that the addresses of the failed nodes were still included as traffic destinations. The position broadcasting and mobility patterns were also replayed using the log files which were created during the real experiment.

In [68], the authors describe three different signal propagation models which they used in their simulations. Their results, however, have shown that only one of these three models matches the real measurements. This propagation model consists of an exponential path loss combined with random fading. They chose 2.8 as path loss exponent and 6 dB as the standard deviation for the random fading.

Wireless network emulation setup

In order to repeat the abovementioned experiment using wireless network emulation, the whole setup had to be mapped to the available tools and mechanisms: ns-3 as network simulation, at least responsible for the wireless channel, and real systems to run the implementation under test. Here, the network simulation was only used to model the wireless channel and Xen paravirtualized machines were used to run all software of the wireless stations.

The network simulation modeled all nodes of the experiment as gateway nodes. Thus, 33 nodes with a WifiNetDevice and a `WifiEmuBridge` were installed in the simulation. The wireless channel was configured to use ns-3's *LogDistancePropagationLossModel* in combination with the *RandomPropagationLossModel*, using the abovementioned parameters of 2.8 as path loss exponent and 6 dB as standard deviation of a normal distributed random variable. The transmission power of the wireless network cards was set to 15 dBm, which is the transmission power of the Lucent Orinoco Wi-Fi cards which were used in the real experiment. The simulated 802.11b network cards were set to ad hoc mode with a fixed data rate of 2Mbit/s. The fact that ns-3 does not implement a full ad hoc mode was an advantage for this setup, as ns-3's ad hoc mode is similar to the demo mode which was used in the real experiment.

While the aforementioned settings were only a matter of configuration options, replaying the correct mobility patterns of the 33 nodes was more complicated: Part of the real experiment's data are GPS log files containing each node's position updated with an interval of 1 second. In order to replay these traces in ns-3, the position information of the log files was in a first step converted to a rectangular grid. These trace files were then read during simulation setup and fed into ns-3's *WaypointMobilityModel*, which takes a list of time/position tuples for each node and interpolates linearly between them.

The remaining components of the emulated nodes were realized through the use of virtual machines. For each emulated node, one PVM with 128 MB of RAM, 4 GB hard disk space and Debian Lenny as operating system was set up. The wireless emulation network card driver provided a wireless network interface for each of the virtual machines and connected to the corresponding gateway node in the network simulation. Using this setup, the virtual machines provided an environment similar to the one of the laptop computers used in the real experiment.

The software run during the outdoor experiment consisted of the ad hoc network routing daemons and a traffic generator. Since the routing daemon software is available as part of the data provided in the CRAWDAD repositories, it could be used in the virtual machines. However, a minor modification was necessary to run this software: The original software used a tunnel device to accept application traffic which is sent through the ad hoc network. The authors of [34] ported this device driver from FreeBSD, whereas the Linux tun device was used in this emulation setup. The traffic generator used in the real experiment is not provided with the experiment's data, for which reason it was rewritten: It simply parses the corresponding traffic log file and sends an UDP message of the logged size to the corresponding destination at the correct point in time.

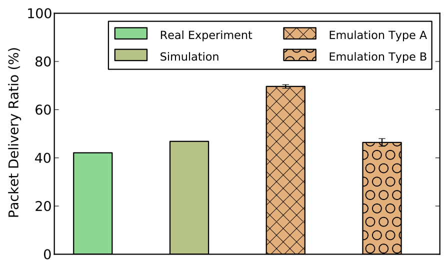
Using the network simulation and the virtual machines, the emulation setup is in principle complete. A last difficulty was to set up all these components in a synchronized manner: The replayed mobility traces used in the network simulation have to match the application traffic generated inside the virtual machines. This was realized using the following mechanism: As part of the simulation run, the network simulation established a control connection to the virtual machines, through which it triggered the setup of the Wi-Fi driver, started the routing software and executed the traffic generator at specified points of the simulation time.

One particular detail of the outdoor experiment, however, could not be repeated: In the real experiment, the nodes broadcasted their current position using the wireless network. As the available publications do not provide any further information about this broadcast mechanism, it was not possible to implement it, leading to a possibly less congested wireless channel.

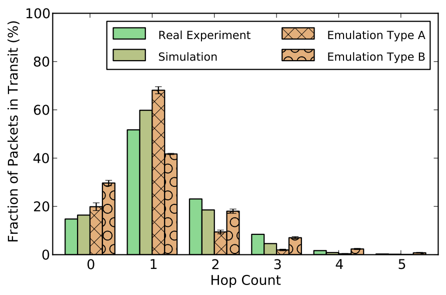
This complete setup was executed on one of the two testing machines. In a first attempt, the simulation was started using the real-time scheduler of ns-3. Although the machine seemed to cope well with the load of the 33 virtual machines, the network simulation could not keep up with the real-time execution: At the moment in which all emulated network card drivers where loaded and registered at the emulation bridges, the simulation stopped because it could not process events fast enough. As a solution, the synchronized network emulation implementation was used with a synchronization time-slice size of 500 µs. Since the synchronized virtual machines are executed for each interval one in a row, the execution was slowed down with a factor of 33, leading to a total execution time of 9 hours for one simulation run. Although this is certainly a drawback of the emulation setup, it enabled the execution of this large setup on a single machine.

Emulation results

Two metrics were used in [68] to compare the real experiment with the simulation results: The packet delivery ratio and a hop-count histogram. The packet delivery ratio is the number of application data packets which could be delivered successfully to its recipient. The hop-count denotes the number of hops after which a packet was either dropped or received by its recipient. However, the hop-count histogram is only included for AODV, which is one of the four routing protocols which were evaluated in the original study. In order to make use of both comparison metrics, only the AODV protocol was used for the emulation experiment.



(a) Delivery Ratio



(b) Hop Count

Figure 5.10 The delivery ratio and hop count measured in the ad hoc outdoor scenario. (error bars indicate the standard deviation, experiment and simulation data from [20, 68])

Figure 5.10 shows these two diagrams using data of the real experiment[34], the simulation results presented in [68] and the emulation results using two different wireless channel settings. Just as the network simulation [68], the network emulation experiment was run five times and the results were averaged. The results denoted as *Type A* have been acquired with the simulation setup described above. The *Type B* results have been acquired using a small variation in the network simulation setup: The thresholds for the necessary received signal power have been increased from ns-3's default values to those which have been found as default settings in the SWAN simulator [98]. Thus, a higher signal strength is necessary in order to receive data. As these settings are not described in the available publications, it is uncertain whether they have been used in the network simulation.

Figure 5.10(a) shows that the delivery ratio of the pure simulation matches the result of the real experiment well. The delivery ratio obtained using ns-3's default signal reception threshold is more than 20 % higher than that obtained from the real experiment and simulation. After increasing the received signal strength threshold, the packet delivery ratio matches the original simulation results.

The results for the hopcount do not match as good as those of the delivery ratio. For emulation type A, the low threshold for packet reception led to a too high number of packets which were transmitted in one hop. As a consequence, there are too few packets which have been routed across multiple hops. For a higher reception threshold, the number of packets transmitted across one hop decreased too much and too many packets could not be routed at all (denoted as zero hops). Although these results do not match the real experiment as good as the results of the pure simulation, they nevertheless show the overall form of the hopcount histogram.

More interesting than the immediate results shown in Figure 5.10 is the experience gained from this emulation setup: The effort required to initially setup this mid-sized emulation scenario is not negligible. However, it was certainly easier than to set up and conduct the real experiment as it is presented in [34]. The other alternative, a pure network simulation, is slightly easier to handle as all software runs in one central place, but has the drawback that the software under test has to be ported to the network simulation. In the case of [68], the authors report that it was relatively easy to convert the ad hoc routing daemons to network simulation code. However,

```
root@wifi-test2:~/wifi-emu-kern# insmod ./wifi-emu.ko client_id=1 && insmod ./wi
fi-emu-udp.ko peer_addr=127.0.0.1
root@wifi-test2:~/wifi-emu-kern# ifconfig wemu0
wemu0     Link encap:Ethernet  HWaddr 00:00:00:00:00:01
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@wifi-test2:~/wifi-emu-kern# iwconfig wemu0
wemu0     IEEE 802.11b  ESSID:"wifi-b"  Nickname:"wifi-emu"
          Mode:Master  Frequency:2.447 MHz  Access Point: 00:00:00:00:00:02
          Bit Rate:1 Mb/s
          Link Quality=45/100  Signal level=-56 dBm  Noise level=-101 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0

root@wifi-test2:~/wifi-emu-kern# iwconfig wemu0 mode Monitor
root@wifi-test2:~/wifi-emu-kern# ifconfig wemu0 up
```

Figure 5.11 Terminal output showing how to load and configure the wireless emulation driver for use in Monitor mode.

the ad hoc routing daemons were already programmed as event-driven software, so that a much higher effort can be expected in general.

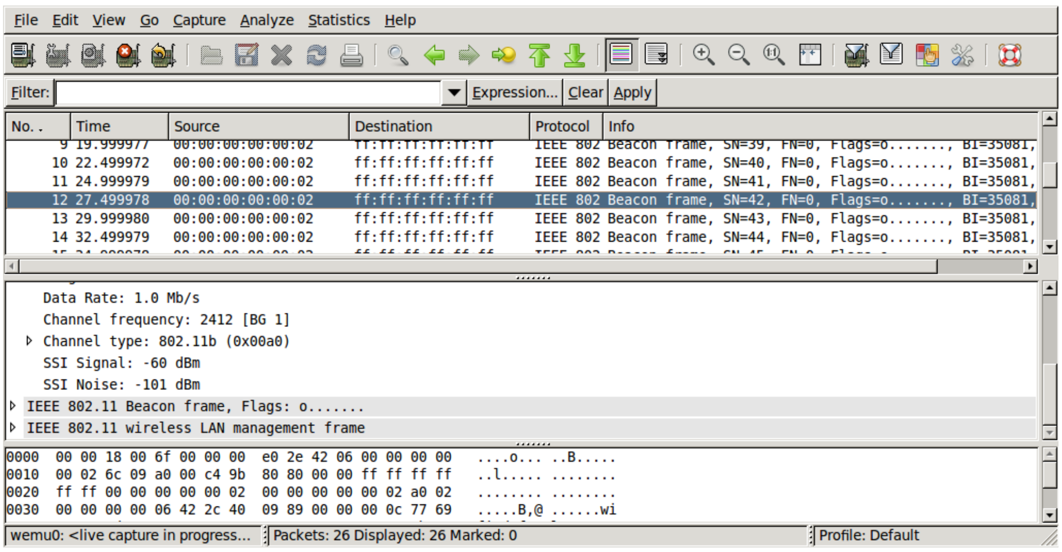
The fact that the emulation results denoted as type A and type B differ that much shows furthermore that an accurate wireless network model is inevitable. Similar experiences have been reported by Liu et al. for their network simulation [68] resembling the real experiment.

5.3 Summary

The results presented in the previous sections gave some insights into several aspects of the emulation environment. This section concludes the evaluation chapter and discusses whether the system fulfills its purpose.

One important factor is that the system is easy to use and flexible. While the previous sections only described how to actually use the system in a very abstract way, Figures 5.11 and 5.12 give a more practical example of a possible usage scenario: The wireless emulation driver is loaded into the system and connects to the network simulation using a localhost network connection. Immediately after this, the wireless network interface is installed in the system and ready for use. As the output of *ifconfig* and *iwconfig* shows, the created interface provides all kind of information which is also provided by real wireless network cards. Afterwards, the device is configured to use the monitor mode and activated. The Screenshot in Figure 5.12 shows how the *Wireshark* [116] protocol analyzer is then used to record the raw 802.11 frames which are received by the wireless network card.

As the above example shows, all that is required for the system under test is to load the wireless network card driver. As the communication with the network simulation takes place via a network connection, the wireless network card driver can be used in any Linux system which supports the loading of additional kernel modules and has some existing network interface which can be used to connect to the simulation.



```
No. .  Time       Source             Destination        Protocol  Info
    10 22.499972  00:00:00:00:00:02  ff:ff:ff:ff:ff:ff  IEEE 802 Beacon frame, SN=40, FN=0, Flags=o......., BI=35081,
    11 24.999979  00:00:00:00:00:02  ff:ff:ff:ff:ff:ff  IEEE 802 Beacon frame, SN=41, FN=0, Flags=o......., BI=35081,
    12 27.499978  00:00:00:00:00:02  ff:ff:ff:ff:ff:ff  IEEE 802 Beacon frame, SN=42, FN=0, Flags=o......., BI=35081,
    13 29.999980  00:00:00:00:00:02  ff:ff:ff:ff:ff:ff  IEEE 802 Beacon frame, SN=43, FN=0, Flags=o......., BI=35081,
    14 32.499979  00:00:00:00:00:02  ff:ff:ff:ff:ff:ff  IEEE 802 Beacon frame, SN=44, FN=0, Flags=o......., BI=35081,

    Data Rate: 1.0 Mb/s
    Channel frequency: 2412 [BG 1]
  ▷ Channel type: 802.11b (0x00a0)
    SSI Signal: -60 dBm
    SSI Noise: -101 dBm
▷ IEEE 802.11 Beacon frame, Flags: o.......
▷ IEEE 802.11 wireless LAN management frame

0000  00 00 18 00 6f 00 00 00  e0 2e 42 06 00 00 00 00   ....o... ..B.....
0010  00 02 6c 09 a0 00 c4 9b  80 80 00 00 ff ff ff ff   ..l..... ........
0020  ff ff 00 00 00 00 00 02  00 00 00 00 00 02 a0 02   ........ ........
0030  00 00 00 00 06 42 2c 40  09 89 00 00 00 0c 77 69   .....B,@ ......wi

wemu0: <live capture in progress...  Packets: 26 Displayed: 26 Marked: 0  Profile: Default
```

Figure 5.12 Screenshot of Wireshark capturing raw 802.11 frames in Monitor mode.

The Linux kernel module has been tested for the Linux kernel versions 2.6.18-xen, 2.6.24, 2.6.26 and 2.6.32.

Starting the simulation and loading the wireless network card driver is in principle everything that has to be done in order to use the emulation environment. However, the outdoor ad hoc network scenario of Section 5.2 has shown that this can be a quite complicated task when several different components are involved in the overall setup. The alternative, the realization of a corresponding wireless network experiment using real systems, is certainly not easier in such cases.

The outdoor experiment has furthermore shown that the emulation system is scalable. Using only a single computer, a wireless network consisting of 33 nodes has been emulated. Even if a single computer does not provide enough computational resources, the emulation environment can be used without any changes across multiple machines. Another point the outdoor experiment has shown is that it is possible to use the same software which is used on real systems as well.

One of the goals defined in Section 3.1 is to provide a realistic wireless environment to the systems under test. Both the basic measurements in Section 5.1.1 and the outdoor ad hoc network study have shown that the emulated wireless networks approximately match the behavior of real wireless networks. In order to improve the realism of the wireless channel, some fine-tuning of the simulation models has to be performed. This drawback, however, is not specific for this wireless emulation environment, but exists in regular network simulations as well.

The behavior of the wireless channel also influences the transparency of the system, meaning whether a software running in emulated systems is under the impression that it uses a native wireless network. Regarding the wireless network itself, it has been shown that the use of virtual machines has no influence on the transparency. Another factor is the functionality which is provided by the emulated wireless network card. In its current state, some functionality like the scanning for available networks is missing, so that a system using these features will realize that it does not use a full-featured wireless network card. It is, however, not possible to define a generic set of required functionality, since different wireless network cards and drivers implement a different set of functions. Up to a certain degree, it is therefore acceptable that the emulated wireless network card does not provide all possible functions.

6 Related Work

This chapter gives an overview of related work in the field of wireless network emulation. As only a handful of approaches exist whose concept is similar to the one proposed by this thesis, wireless network emulation is presented in a broader form: What is meant by wireless network emulation in this chapter is the utilization of system in which one or more of the wireless network related layers have been replaced or altered in some way. The higher layers of the communication systems, however, use unmodified software like it would be used in a real experiment.

Section 6.1 introduces a number of solutions which make use of real wireless network hardware, but nevertheless differ from classical experiments. The approaches presented in Section 6.2 are pure software solutions building upon a custom emulation engine. Finally, Section 6.3 introduces approaches which utilize network simulations and are therefore most similar to the concept proposed in this thesis.

6.1 Hardware Emulation

The approaches presented in this section differ from classical experiments in that they automate parts of the experiment and thereby ease the process of evaluation.

The *Ad hoc Protocol Evaluation* (APE) testbed [77] is a software framework which is intended to be used in classical experiments where mobility is achieved through participating persons which carry laptops. One major concern in such experiments is that it is almost impossible to repeat an experiment in the same manner. In order to address this concern, APE supports the experiment execution in a number of ways: A *scenario file* contains information at which point in time certain actions shall happen. These can be commands which are executed or information which is displayed to the test participants, such as instructions to move to a certain point. It furthermore contains logging mechanisms, which for example store the signal strength of neighboring nodes in order to create a connectivity map after the experiment. As APE does not modify the wireless network itself, it does, strictly seen, not belong to the class of wireless network emulation solutions, but is something in between classical experiments and network emulation.

Another issue in wireless network experiments is the huge requirement of space. As the communication range of a regular wireless network can be up to a few hundred meters, experiments have to be performed stretched over large areas. *Testbed on a Desktop* [56] approaches this problem through the use of signal attenuators which are mounted between the antenna connectors of wireless network cards and antennas. Thereby, the communication range of the wireless network cards is reduced, so that larger topologies can be evaluated within in a single room. By using different attenuators and through the physical placement of antennas, different wireless network topologies can be evaluated.

The *Emulated Wireless Ad Hoc Network Testbed* (EWANT) [91] continues this approach. Through the use of computer-switchable multiplexers which are connected to the antenna ports, different antennas with different characteristics can be used at a single wireless network card. By switching the antennas used in the course of an experiment, it is therefore possible to create a change in signal reception, just as it normally happens through mobility of stations.

A more sophisticated approach using attenuated wireless signals can be found in the *Miniaturized Network Testbed for Mobile Wireless Research* (MiNT) [21]. Similarly to the abovementioned projects, attenuated signals are used to decrease the range of wireless network cards. In order to create mobility patterns, the antennas are mounted on mobile robots whose movement can be controlled centrally. Furthermore, MiNT includes software to graphically configure an experiment and to collect packet traces during the experiment. However, only the antennas are mounted on the mobile robots and are connected via cables to the wireless network cards installed in Desktop PCs. As this puts additional constraints on the possible mobility patterns, the successor project MiNT-2 [74] uses embedded computers which are completely mounted on a mobile platform. MiNT-2 also encloses further improvements, like the use of RFID tags to facilitate the localization of mobile stations.

A technique providing a very high level of control over the wireless channel is proposed in [55]. This approach, which the authors call *physical emulation*, is often referred to as the *CMU Emulator*. It also makes use of regular stations which are equipped with wireless network cards. The antenna ports of the network cards are, however, not connected to antennas, but to an emulator which consists of one or more FPGAs. The RF signals which are transmitted by the wireless network cards are mixed down to a baseband signal and digitized. The FPGA models various effects of wireless signal propagation and converts the resulting output signals back to RF signals which are fed into the corresponding antenna ports of the stations. The FPGAs are controlled by a software which models node movement and other effects.

Compared to the emulation environment which was developed within this thesis, the emulation approaches described above provide a very authentic environment to the software under test, since they make use of complete systems which include a physical wireless network card. This is also their major drawback: They require a single computer for each emulated station and in some cases additional hardware.

A further drawback of the approaches using signal attenuation is that they allow mobility only within certain limits, which is bounded through the attenuators used and the size of the room in which the experiment takes place.

6.2 Software Emulation

The approaches described above require a dedicated physical machine equipped with wireless network hardware for each station of the emulated network. This necessity is due to the fact that the wireless network is directly emulated at the physical layer. The solutions presented in this section perform the wireless network emulation at higher layers of the network stack and thereby eliminates the need for wireless network hardware.

Modified local area networks

One way to achieve the emulation of wireless networks is the use of wired local area networks which have been altered in order to resemble the behavior of a wireless network.

An early approach of this kind is presented in [76]. The idea of this approach is to create a wireless network model out of traces which have been collected in a real experiment. In concrete terms, this means that during one or more runs of an experiment, characteristics of a wireless network are measured and collected. Later-on, the experiment can be replayed in an emulated network. This network consists of regular computers which are connected through a wired LAN. The operating system kernel executed on these stations is modified to drop and delay network packets according to the network model which has been created from the experiment data. Thereby, it is possible to repeat experiments in a network with similar characteristics to the one in which the measurements have been taken.

Another solution for wireless network emulation is *MobiEmu* [120]. It uses ns-2 scenario files to generate connectivity rules which determine the stations that are within reception range of each other at a certain point in time. Similarly to above-mentioned solution, all stations used for emulation are physically connected using a wired LAN. Based on the rules generated beforehand, firewall rules are inserted and removed during runtime. These rules drop the received network packets of stations which are currently out of reception range. In order to ensure that the rules are inserted and removed in a synchronous manner, a master controller broadcasts the current time whenever changes in the network topology happen. As an alternative to single physical stations, MobiEmu also supports the virtualization of stations through the use of User Mode Linux [22].

A similar approach is used in [85]. Here, dummynet [89] is used to put constraints like bandwidth limitations or delay on the IP traffic of an emulated network. The parameters for the network traffic are generated through the use of a discrete event simulation. This custom network simulation is run before the actual emulation itself starts and computes the constraints which are to be put on the emulated network.

The three aforementioned approaches have in common that the wireless network properties are precomputed before the actual emulation takes place. The advantage of executing these computations in a separate step is that there are no real-time constraints, so that arbitrarily complex propagation models can be used to compute the wireless network parameters. The approach which this thesis proposes suffers in principal also from these real-time constraints. Through the use of synchronized network emulation, as used in Section 5.2, the real-time constraints can be circumvented.

Another drawback of the solutions described above is the restricted manner in which the network traffic is altered. Simple rules which drop, delay or limit the throughput of network packets can resemble the behavior of a wireless medium only in a limited way. Furthermore, the set of available actions is fixed and a change in this regard requires the adaption of both the wireless network precomputation and the network emulation itself. Through the use of a network simulation, as it is proposed by this thesis, the emulated network traffic is altered inside the simulation, which allows to model the wireless channel at any desired level of detail.

Complete emulation

The abovementioned solutions have in common that they make use of complete systems which have been modified in order to act like a wireless network. In contrast to this, approaches exist which emulate the wireless network completely on their own. Thereby they are, at least from a conceptual point of view, not restricted in which way emulated network traffic can be modeled.

One of these approaches is *NEMAN* [84], an emulation tool for wireless ad hoc networks. It uses a software called *topology manager*, which creates a tap-interface for each emulated station. Using these interfaces, the software under test is instantiated once for each interface. Whenever a network frame is sent through one of these interfaces, the topology manager forwards this frame to all interfaces which correspond to stations that are within reception-range of the sender. As the creation of virtual network interfaces is very lightweight, it is possible to emulate hundreds of nodes on a single machine. This approach, however, also has its drawbacks: The applications using the tap-interfaces have to be configured to specifically use the according network interface, which can require a recompilation of the software. Furthermore, the kernel does not support local communication through its own interfaces, for which reason the Linux kernel has to be modified. Another constraint is that routing cannot happen through the Linux kernel, but has to happen inside the topology manager. This requires a change in the corresponding ad hoc network routing daemon. As a consequence, it is not possible to run unmodified application software, for which reason this approach becomes less transparent.

Another network emulation approach which aims at the emulation of large wireless networks is *CORE* [4]. It is built upon several virtualization mechanisms which are provided by the FreeBSD [101] kernel. Each station of a wireless network is run in a virtual execution environment that has its own virtual network stack. These stations are connected to each other using virtual network equipment which is also provided by FreeBSD. This interconnecting network can be created and configured through a graphical user interface. With regard to wireless networks, CORE supports two modes: A simple on-off mode can be used to enable and disable the packet reception depending on the reception range. Additionally, external programs can be plugged in to emulate more complex wireless network behavior: In this mode, the graphical user interface sends information about the nodes to the daemon, which can configure properties such as delay, loss or bandwidth limitations.

MobiNet [72] is a software for the emulation of wireless ad hoc networks. With this approach, only a certain number of nodes are emulated through the use of real systems. These so-called *edge nodes* are connected to MobiNet, which runs as part of the FreeBSD kernel. The ad hoc network through which these nodes are connected is completely emulated by MobiNet on its own. A mobility model generates mobility patterns and loads them into MobiNet. When a packet enters the MobiNet core, a routing module tries to find a path to the destination. In [72] a dynamic source routing module is used for this purpose, but additional routing protocols could be implemented as well. Using the generated route information, a MAC-layer module performs the actual network modeling itself and emulates how a packet is sent across the selected connections and stations. When the packet reaches its destination, it is delivered to the corresponding edge node system. MobiNet therefore differs conceptually from the other approaches which have been presented in this section in that the ad hoc network implementation is part of the emulation framework and is not executed as part of the real systems. While this certainly requires less resources, it is an additional restriction that routing protocols have to be implemented as part of the MobiNet core.

A similarity between these systems is that they do not require a full station for each station of the network. While this allows for a very efficient execution of the complete emulation setup, it restricts the software under test: With NEMAN, software has to be modified and MobiNet requires the ad hoc network routing to be implemented as part of MobiNet itself. The lightweight execution environments utilized by CORE are restricted in that they cannot be used to execute arbitrary software, but can be replaced through full systems.

The approach which this thesis proposes is more flexible in this regard. As a wireless network driver and an existing network connection are the only requirements, the emulated network card can be used in a natively executed operating system as well as different kinds of virtualization environments. In its current implementation for Linux, it is naturally restricted to Linux as operating system for emulated stations. However, the concept itself allows to implement the driver for other operating systems as well.

A very simple system called *hwsim* [71] is part of the Linux kernel. It consists of a kernel module that installs a number of 802.11 wireless network devices that are connected to each other. However, it does not contain any physical layer model and simply copies sent frames to all stations that use the same frequency. It is intended to be used for simple functional testing of software which uses the wireless network interfaces of Linux. Thereby, it differs from the other approaches which have been presented in this section.

6.3 Emulation through Simulation

Another way to perform wireless network emulation is to utilize a network simulation which emulates the wireless channel. This is the technique which is also used by the concept which this thesis proposes.

JiST/MobNet

Another approach which utilizes network simulations is *JiST/MobNet* [62]. It is based on the JiST [9] simulator and its SWANS ad hoc network simulation models.

The concept used in JiST/MobNet differs slightly from the one of this thesis: It is not only supposed to be used for wireless network emulation, but for simulation and real-world experiments as well. Thereby, it shall be possible to use the same tools across all three evaluation methodologies.

As JiST itself does not provide support for network emulation, the authors had to extend it in several ways. Besides a real-time kernel which allows to execute simulations in real-time, two different network emulation devices were implemented: A *tun wrapper* and a *pcap wrapper*.

The tun wrapper is supposed to be used for wireless network emulation. It creates a tunnel device[1] through which applications running in the real system can exchange network packets with JiST/MobNet and are thereby integrated into the simulation.

The pcap wrapper works in the opposite way and connects to an already existing network device. It uses the widespread *packet capture library* (libpcap) [100] to exchange Ethernet frames with network devices like a wireless network card. Its purpose is to perform real-world experiments which use a real wireless network connection in combination with JiST/MobNet.

As the primary focus of this thesis is the evaluation of real software in emulated wireless networks, a comparison with the emulation capabilities of JiST/MobNet is most reasonable. Just like the wireless network card driver developed within this thesis, the tun wrapper provided by JiST/MobNet provides a virtual network device which can be used by real implementations to access an emulated wireless network. However, the interface between real systems and simulation differs slightly. While the concept of this thesis is to provide an interface that acts like a wireless network card, JiST/MobNet provides tun-devices which exchange IP packets instead of Ethernet frames with the real system. Another difference is that JiST/MobNet does not offer any wireless-specific functionality through this interface. Thus, an application which runs in the real system can send and receive network packets through this interface, but cannot access wireless-specific functionality.

A further difference is that the tun-devices can only be created inside the system in which the network simulation is running. This restriction to a single system has several consequences: Likewise to NEMAN, all instances of the software under test have to be started on a single system, which can lead to problems as certain networking functionality is not available for this form of local communication. Furthermore,

[1]The so-called tun-devices are similar to tap-devices, but exchange IP packets instead of Ethernet frames.

the use of a single system means that all instances of the software under test make use of the same network stack, which can result in behavior that is different from the use of individual network stacks for each system. Another consequence is that it is not possible to spread the load of network simulation and software under test over several systems, which restricts the scalability of this approach.

The pcap wrapper which is provided by JiST/MobNet is intended to be used for real-world experiments in which existing wireless network devices are used. Clearly, the emulation environment developed within this thesis cannot provide this functionality as it is supposed to be used in the opposite direction. However, the EmuNetDevice of ns-3 (cf. Section 2.5.2) provides this functionality, so that ns-3 could be used for such experiments as well.

ns-2 extension

Another implementation for wireless network emulation is described in [93, 94]. The focus of this work is to ease the evaluation of a certain aspect in *radio access networks*: When distributed applications exchange data using wireless communication by means of network access, changes between wireless networks cause address changes which in turn influence the data transport of these applications.

As a methodology for the evaluation of the behavior of distributed applications under such circumstances, network emulation using ns-2 is proposed. The already existing infrastructure of ns-2 was extended to be able to deal with dynamic address changes and roaming between networks. To that end, a protocol called INFRA was developed. It does not model the corresponding processes of real networks exactly, but imitates its behavior which consists of beacon messages being sent by an access point, network association and address distribution. The address distribution mechanism resembles the one of DHCP [23], but is initiated through beacons and not through a probe-request by the client.

The actual emulation takes place through a *virtual tunnel* (vtun) device which is created by the simulation. Dynamic address changes inside the simulation are also exported to the tun-device, so that applications have to cope with the same effects of changing addresses as in real wireless networks. A further extension of this device, called *wtun*, implements the wireless extensions of Linux and thereby exports properties of the wireless network to applications. The simulation pushes the corresponding information in a certain interval (by default 0.5 s) to the wtun device, so that all information is directly accessible whenever the device is queried using the wireless extensions.

Compared to the emulation environment which was developed within this thesis, the wtun device offers similar capabilities: Real systems can access the emulated network through a virtual wireless network device. However, all wtun devices are instantiated in the same system, which can lead to problems as described above for NEMAN and JiST/MobNet.

A further difference is the network layer at which the integration of real systems and simulation takes place. The emulation environment developed within this thesis emulates only the physical and data link layer, for which reason all functionality from the network layer upwards is implemented solely through real systems. Scenarios like the aforementioned change of network addresses could therefore be implemented as part of the real systems using regular DHCP implementations, which is not possible with the ns-2 extension described above.

VirtualMesh

The wireless network emulation approach, whose concept is most similar to the one developed in this thesis, is *VirtualMesh* [97]. It is primarily intended for the evaluation of mesh networks [12] and is based on the *OMNeT++* [108] network simulator.

Within the network simulation, a simulated 802.11 network forms the base of the setup. Some of the simulated hardware devices are used as part of so-called *Proxy-Hosts* which act as a proxy between the wireless network and external Linux systems. These systems can be real or virtualized systems. They execute a software called *PacketModeller* which runs in user space and creates a tap-device. Network frames sent and received by this tap-device are exchanged with the network simulation over UDP. Thereby, the tap-interface can be used to connect to the simulated WLAN. In addition to the Ethernet frames which are sent and received, parameters of the wireless network are enclosed within the UDP messages.

The tap-device itself is only used as a regular Ethernet device which does not provide any wireless-specific functionality. An additional program called *virtual interface daemon* (vifd) is used to provide such functionality. vifd is connected to the Packet-Modeller and thereby has access to the parameters of the wireless network. However, programs like the wireless tools [106] would normally use the wireless extensions of Linux to access such information. Therefore, all corresponding software has to be modified to communicate with vifd instead.

Compared to the emulation environment of this thesis, VirtualMesh provides roughly the same functionality. The fact that VirtualMesh uses an unmodified tap-device allows it to run as a user space application, but requires the modification of all software that makes use of the wireless extensions.

In [97], the authors measured the round trip time of ICMP echo messages when using VirtualMesh and XEN paravirtualized machines. They report that VirtualMesh introduces additional delays and measured round trip times of about 3 ms per hop, which is much higher than the real round trip time measurements which have been presented in Section 5.1.1.

7

Future Work

This chapter introduces additional aspects which are worthwhile to be investigated further, but could not be covered within this thesis. To begin with, Section 7.1 outlines three areas through which the already performed evaluation could be extended. Additional functionality which could be added to the implementation is presented in Section 7.2. Furthermore, there are additional fields to which this emulation environment could be applied. Two of them are introduced in Section 7.3.

7.1 Further Evaluation

In addition to the evaluation which has been presented in Chapter 5, there are further aspects which would be worth to investigate. Some of them are described in the following section.

Hybrid scenarios

One aspect which has not been explicitly covered during the evaluation is the capability to perform hybrid emulations, meaning to purely simulate some of the wireless stations while others make use of the emulation mechanism. The experiments in which ICMP echo requests were sent to the simulated access point, made use of this feature in principle, but do not show its full potential. It would, for instance, be interesting to repeat the outdoor emulation experiment of Section 5.2 in a hybrid fashion, so that only a part of the nodes are realized through real implementations and the rest is simulated within ns-3.

Realistic 802.11 model

The measurements of Section 5.1 have shown that the delay which emerges through the use of an emulated 802.11 network connection is composed of several components.

Although the different parts of the emulation environment induce additional delays, the overall delay between two hosts was lower than the one measured using real systems. The delay which is simulated by ns-3 is basically composed of a propagation delay in the PHY layer and the channel access of the MAC layer. In real systems, however, additional delays arise through data processing.

For a better transparency of the emulation environment, it would be good if the delays using wireless network emulation would match those of physical systems more closely. One way to achieve this goal is to introduce additional delays in the simulation model which account for the processing delays that occur in real systems. However, these delays depend on several aspects such as the wireless network hardware or the operating system used, so that a further evaluation is necessary before the simulation model can be modified.

802.11g networks

The measurements which have been performed in Chapter 5 made use of 802.11a and 802.11b wireless networks. By now, both standards are more than 10 years old and their successors 802.11g and 802.11n are widely used. However, the current version of ns-3 only includes models for 802.11a and 802.11b networks, so that it is not possible to use the emulation environment with one of the newer standards. The upcoming version ns-3.10 will contain support for the 802.11g standard [79], so that it will be possible to use the emulation environment for 802.11g networks as well. In order to evaluate how well the new models match the behavior of real networks, it would be useful to repeat the performed measurements for 802.11g.

7.2 Additional Functionality

The functionality provided by the emulation environment influences how well it can be used for the evaluation of certain wireless network scenarios. This section discusses some additional features which could ease the development process in certain cases.

Scanning and configuration

As described in Section 4.3, the emulation bridge inside ns-3 currently does not implement the functionality which is required to scan for available networks or to configure the simulated network card through the driver. As a consequence, it is required to configure the wireless network during simulation setup. While in many cases the associated network is not changed anyway, certain scenarios, such as the handover between wireless networks, exist, in which this missing functionality restricts the applicability of the emulation environment.

For the current implementation, only minor modifications were necessary in order to extract all relevant information from ns-3's Wi-Fi stack. Since the concept of ns-3 is to set up everything before the simulation starts, it is, however, not possible to change all available parameters during runtime. Therefore, it would require major architectural changes if the emulated wireless network card shall be configurable in any possible way.

Nevertheless, some parameters are more easy to change than others, so that a basic version could be implemented relatively easy: Through configuration of the channel number, the SSID and the BSSID it would be possible to determine the network the wireless network card shall be associated with. To support these parameters for master mode would require to adapt a state machine which is used inside the simulation model, but no big changes other than this. To enable the configuration of the transmission bitrate or to change between ad hoc and master mode would be more difficult. The abovementioned more easy changes would therefore only allow to switch between different networks in master mode. Since ns-3 only implements a very limited ad hoc mode, the corresponding functionality for ad hoc mode would not be of much help anyway.

While the aforementioned changes would allow to configure the wireless network, the first step is usually to scan for available networks. ns-3 already implements the corresponding 802.11 messages, but uses them only in such fashion that a probe request for the statically configured network is sent whenever the association with the network is lost. What would rather be required is to send a probe request for all available networks and return a list of them. However, the ns-3 website provides contributed code with this functionality [14] (for an older version of ns-3), which could be used as a starting point.

The wireless network card driver and the corresponding communication protocol already implement all required functionality, so that the aforementioned changes in ns-3 would allow to scan for available networks and to configure the wireless network card. It would be possible through these changes to use the wireless emulation environment for scenarios such as a handover between different networks or localization based on the received signal strength of available networks.

Operating system support

Another factor which restricts the applicability of the emulation environment is the fact that the wireless network card driver is only implemented for Linux. Therefore, it is not possible to use any other operating system to evaluate software in combination with an emulated wireless network.

One solution would be to implement a corresponding wireless network card driver for each operating system which shall be used for the evaluation of wireless network software. Such additional network card drivers could then be used with the existing wireless emulation bridge of ns-3.

As has been mentioned briefly in Section 3.2, an alternative approach would be to emulate the hardware of a wireless network card. Since virtual machines such as Xen HVMs or QEMU [10] implement the hardware of a system in software, it would be possible to emulate a wireless network card which is connected to the network simulation. The only prerequisite of an operating system which runs in such a VM would be that it provides a driver for the wireless network card whose hardware is emulated. Thus, such an implementation of a hardware-emulated wireless network card could be used for different operating systems.

The latter approach is, however, more complex in its implementation as all functionality of the corresponding hardware has to be modeled. For QEMU, and therefore also Xen HVMs[1], an existing patch [61] containing a hardware model for an Atheros wireless network card could serve as a basis. The hardware interfaces which such a model provides inside the virtual machine are only one part of the work, as it would still have to be connected to the wireless network card model of the simulation.

Virtual GPS

An additional feature which would be of use for certain scenarios is the availability of position information. When using the wireless emulation environment, the mobility of stations is modeled within the network simulation. The position of a station is exported implicitly to real implementations through the signal quality, but it is not possible for a station to obtain its current position directly.

A research area in which such functionality would be of use is *location-aided routing* (LAR) [60]. With LAR, the physical position of nodes is utilized for routing-decisions. In order to evaluate implementations of such routing algorithms, it would therefore be useful if stations of an emulated wireless network could obtain their current position according to the chosen mobility model.

A usual way to obtain position information in real experiments is to use the *global positioning system* (GPS) [41]. Likewise, a virtual GPS receiver could be implemented for the wireless emulation environment, so that the software under test could obtain position information. A possible way to implement this would be to extend the wireless network card driver through a serial interface that outputs position data in the widespread NMEA 0183 [66] format. Thereby, software on systems using the emulated wireless network card could obtain position information just as if they would use a real GPS device. The wireless emulation bridge in the network simulation would have to be modified to periodically obtain the node's position from the mobility model and to send it to the associated driver. As the position system used inside ns-3 are simple x-y-z coordinates, the corresponding positions would have to be converted to the latitude/longitude-format which is used by GPS.

7.3 Application Areas

The intended application area of the wireless emulation environment is to ease the development of software using regular 802.11 wireless network cards. Two related areas for which wireless network emulation could also be beneficial are access points and mesh networks.

Access points

Another field to which wireless network emulation could be applied is the development of software for access points. Many access point devices are implemented as

[1]The HVM implementation of Xen makes use of QEMU's hardware models.

embedded systems which run a vendor-specific Linux system. Projects like OpenWRT [25] allow developers to compile their own Linux system for customary access point devices.

As access points are inexpensive and easily modifiable, many research projects use them as development platforms (e.g. [35], [105] or [117]). Therefore, it might ease the development process of such projects if access point software could be tested in an emulated network as well. In principle, it is possible to compile software such as OpenWRT for x86 processors, include the emulated network card driver and run it in a virtual machine. In combination with other stations using the emulated wireless network card, it would then be possible to emulate a complete network, including the software which is intended to be deployed on access points later on.

However, the current implementation of the emulated network card driver does not fully support such a setup. Since access points have to provide different functionality on the MAC layer, the wireless network cards used in real access points have to provide additional functionality or have to allow raw access to the MAC layer. Programs such as hostapd [42] are used on such devices to provide the access point functionality in software, but require a wireless network card driver which provides the abovementioned functionality.

In order to enable emulation support for wireless access points, it is therefore necessary to adapt the emulated network card so that it allows to run software such as hostapd. Since the MAC layer is currently implemented as part of ns-3, this would require to adapt all components of the wireless emulation environment, namely the wireless emulation bridge in ns-3, the wireless network card driver for Linux and the communication protocol used between them.

One possible way to implement such functionality would be to implement the MAC layer as part of the network card driver. The mac80211 [70] library, which is part of the Linux kernel, readily provides the corresponding functionality, so that the MAC functionality itself would not have to be implemented. However, this would require to integrate the simulation and the emulated network card driver at a different layer-boundary, which has several disadvantages compared to the current design (see Section 3.2).

The aforementioned solution using mac80211 is only one possible way. An alternative might also be to leave all MAC functionality in the network simulation and to add additional interfaces to the wireless emulation bridge and the network card driver. Which of the two designs would be better suited has to be evaluated in further research.

802.11s mesh networks

Mesh networks [12], which are multihop ad hoc networks that connect to external networks such as the Internet, are another area to which the wireless emulation environment could be extended. Many mesh network implementations make use of 802.11 network cards which are configured to ad hoc mode and implement the routing functionality in the network layer. For such mesh network implementations, it is already possible to evaluate them using the wireless emulation environment.

The new 802.11s [48, 40, 13, 7] standard[2], however, implements the mesh functionality in the MAC layer, thus allowing the transparent use of any layer-3 protocol. In order to evaluate software which uses 802.11s networks, it might be useful to extend the wireless emulation environment to support this standard.

This goal could be achieved in two different ways. The first one would be the use of 802.11s software implementations like open80211s [80]. This implementation utilizes the abovementioned mac80211 library to add support for 802.11s to regular Wi-Fi cards and would therefore require changes similar to the ones described above for the support of access points.

The other option would be to leave the 802.11s specific functionality inside the network simulation. Since ns-3 already contains models for 802.11s [5], it might be possible to adapt the wireless emulation bridge so that it can be used in conjunction with the mesh network stack. Whether the evaluation of 802.11s mesh networks is worthwhile and which of these designs is better suited has to be evaluated further.

[2]Strictly speaking, 802.11s is not a standard yet, but is in draft status. However, it is quite far progressed and shall be standardized by the end of this year [49].

8

Conclusion

This thesis has presented an approach which facilitates the evaluation of wireless network software. By implementing only the upper layers as part of real systems, it relegates the problematic wireless channel into the controllable network simulation. This is achieved through a wireless network card driver which is used in regular systems that execute the software under test. It is hereby possible to evaluate wireless network software in an authentic environment without having to deal with the otherwise cumbersome evaluation process that wireless networks can cause.

The evaluation results indicate that the developed system is accurate enough to approximately reproduce the behavior of real wireless networks. However, small discrepancies in the measurements of real networks and emulated networks show that the underlying simulation models do not provide an exact representation of a real wireless network implementation. In oder to achieve well-matching emulation results it might therefore be required to fine-tune the simulation models and further investigate these issues.

Although the use of simulation models leads to the aforementioned slight inaccuracies, they are a crucial point as they allow by design to repeat wireless measurements in an easy and controllable manner. A further point in this regard is the scalability of the system. Through the use of virtualization, hardware requirements can be reduced to a minimum: Using only a single computer, it is possible to evaluate a handful wireless network stations in an emulated wireless network.

While this is already a low hardware requirement compared to real experiments, it does not reach the scalability of regular network simulations. Through the use of hybrid emulation setups, it is, however, possible to combine the best of both worlds in one network: An efficient implementation of a large quantity of simulated nodes, with the possibility to evaluate unmodified software prototypes in full-featured operating systems.

The developed system is also very flexible with regard to the use of real systems. Through the utilization of an existing network connection, it is possible to use the developed wireless network card driver on one local machine together with the network simulation, distributed across several computers or inside arbitrary virtualization environments.

All in all, the developed emulation environment allows the flexible evaluation of unmodified wireless network software in a nevertheless realistic and scalable wireless network environment.

Bibliography

[1] Abramson, N. THE ALOHA SYSTEM: another alternative for computer communications. In *AFIPS '70 (Fall): Proceedings of the November 17-19, 1970, fall joint computer conference* (New York, NY, USA, 1970), ACM, pp. 281–285.

[2] Aguayo, D., Bicket, J., Biswas, S., and De Couto, D. MIT roofnet: Construction of a production quality ad-hoc network. MobiCom Poster, 2003.

[3] Ahn, J., Danzig, P., Liu, Z., and Yan, L. Evaluation of TCP Vegas: emulation and experiment. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (1995), ACM, p. 195.

[4] Ahrenholz, J., Danilov, C., Henderson, T., Kim, J., and Works, B. Core: A real-time network emulator. In *IEEE Military Communications Conference, 2008. MILCOM 2008* (2008), pp. 1–7.

[5] Andreev, K. IEEE 802.11s Mesh Networking NS-3 Model. Talk given during the "Workshop on ns-3", March 15th, 2010, Malaga, Spain. `http://www.nsnam.org/workshops/wns3-2010/presentation.pdf`, 3 2010. Accessed June 30, 2010.

[6] Atheros. White paper – 802.11 wireless lan performance. `http://www.atheros.com/whitepapers/atheros_range_whitepaper.pdf`, 4 2003. Accessed May 23, 2010.

[7] Bahr, M. Update on the hybrid wireless mesh protocol of IEEE 802.11 s. In *IEEE Conference on Mobile Adhoc and Sensor Systems* (2007).

[8] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003), ACM, p. 177.

[9] Barr, R., Haas, Z., and van Renesse, R. JiST: An efficient approach to simulation using virtual machines. *Software: Practice and Experience 35*, 6 (2005), 539–576.

[10] Bellard, F. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference* (2005), USENIX.

[11] Bless, R., and Doll, M. Integration of the FreeBSD TCP/IP-stack into the discrete event simulator OMNet++. In *WSC '04: Proceedings of the 36th conference on Winter simulation* (2004), Winter Simulation Conference, pp. 1556–1561.

[12] Bruno, R., Conti, M., Gregori, E., et al. Mesh networks: commodity multihop ad hoc networks. *IEEE Communications Magazine 43*, 3 (2005), 123–131.

[13] Camp, J., and Knightly, E. The IEEE 802.11 s extended service set mesh networking standard. *IEEE Communications Magazine 46*, 8 (2008), 120–126.

[14] Carneiro, G. experimental wifi scanning. `http://www.nsnam.org/contributed/ns-3-wifi-scanning.tar.bz2`. Accessed July 1, 2010.

[15] Cerf, V., and Kahn, R. A protocol for packet network intercommunication. *IEEE Transactions on Communications 22*, 5 (1974), 637–648.

[16] cfg80211 – Linux Wireless. `http://wireless.kernel.org/en/developers/Documentation/cfg80211`. Accessed May 10, 2010.

[17] Chisnall, D. *The Definitive Guide to the Xen Hypervisor*, 1 ed. Prentice Hall, 12 2007.

[18] Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., and Bowman, M. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev. 33*, 3 (2003), 3–12.

[19] Corbet, J., Rubini, A., and Kroah-Hartma, G. *Linux Device Drivers*, 3 ed. O'Reilly Media, 2005.

[20] Crawdad metadata: dartmouth/outdoor. `http://crawdad.cs.dartmouth.edu/meta.php?name=dartmouth/outdoor`. Accessed May 16, 2010.

[21] De, P., Raniwala, A., Sharma, S., and Chiueh, T. MiNT: a miniaturized network testbed for mobile wireless research. In *Proc. IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM 2005* (2005), vol. 4, pp. 2731–2742 vol. 4.

[22] Dike, J. A user-mode port of the linux kernel. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference* (Berkeley, CA, USA, 2000), USENIX Association, pp. 7–7.

[23] Droms, R. RFC 2131: Dynamic host configuration protocol. The Internet Engineering Task Force, 1997.

[24] Erazo, M. A., Li, Y., and Liu, J. SVEET! a scalable virtualized evaluation environment for TCP. In *Proc. 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops TridentCom 2009* (2009), pp. 1–10.

[25] Fainelli, F. The OpenWrt embedded development framework. In *Proceedings of the Free and Open Source Software Developers European Meeting* (2008).

[26] Fall, K. Network emulation in the Vint/NS simulator. In *IEEE International Symposium on Computers and Communications, 1999. Proceedings* (1999), pp. 244–250.

[27] Fazel, K., and Kaiser, S. *Multi-carrier and spread spectrum systems.* Wiley, 2003.

[28] Fishman, G. S. *Principles of Discrete Event Simulation.* John Wiley & Sons, Inc., New York, NY, USA, 1978.

[29] Fujimoto, R. M. Parallel discrete event simulation. In *WSC '89: Proceedings of the 21st conference on Winter simulation* (New York, NY, USA, 1989), ACM, pp. 19–28.

[30] Gast, M. S. *802.11 Wireless Networks: The Definitive Guide*, 2 ed. O'Reilly Media, 5 2005.

[31] Glossary – Linux Wireless. `http://linuxwireless.org/en/developers/Documentation/Glossary`. Accessed April 7, 2010.

[32] Goldberg, R. Survey of virtual machine research. *IEEE Computer 7*, 6 (1974), 34–45.

[33] Grau, A., Maier, S., Herrmann, K., and Rothermel, K. Time Jails: A hybrid approach to scalable network emulation. In *Proc. 22nd Workshop on Principles of Advanced and Distributed Simulation PADS '08* (2008), pp. 7–14.

[34] Gray, R. S., Kotz, D., Newport, C., Dubrovsky, N., Fiske, A., Liu, J., Masone, C., McGrath, S., and Yuan, Y. Outdoor experimental comparison of four ad hoc routing algorithms. In *Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)* (2004).

[35] Heer, T., Li, S., and Wehrle, K. PISA: P2P Wi-Fi Internet Sharing Architecture. In *Proceedings of the Seventh IEEE International Conference on Peer-to-Peer Computing* (2007), IEEE Computer Society, pp. 251–252.

[36] Heidemann, J., Bulusu, N., Elson, J., Intanagonwiwat, C., Lan, K., Xu, Y., Ye, W., Estrin, D., and Govindan, R. Effects of detail in wireless network simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation* (2001), pp. 3–11.

[37] Henderson, T., Lacage, M., Riley, G., Dowell, C., and Kopena, J. Network simulations with the ns-3 simulator. In *SIGCOMM Demonstration* (2008).

[38] Henderson, T., Roy, S., Floyd, S., and Riley, G. ns-3 project goals. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator* (2006), ACM, p. 13.

[39] Hiertz, G., Denteneer, D., Stibor, L., Zang, Y., Costa, X., and Walke, B. The IEEE 802.11 universe. *Communications Magazine, IEEE 48*, 1 (january 2010), 62 –70.

[40] Hiertz, G., Max, S., Zhao, R., Denteneer, D., and Berlemann, L. Principles of IEEE 802.11 s. In *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on* (2007), pp. 1002–1007.

[41] Hofmann-Wellenhof, B., Lichtenegger, H., and Collins, J. *Global positioning System. Theory and Practice.* Springer, Wien (Austria), 1993.

[42] hostapd: Ieee 802.11 ap, ieee 802.1x/wpa/wpa2/eap/radius authenticator. `http://hostap.epitest.fi/hostapd/`. Accessed July 1, 2010.

[43] IANA. Ether Types. `http://www.iana.org/assignments/ethernet-numbers`. Accessed March 18, 2010.

[44] IEEE, Ed. *802.2-1998 IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements–Part 2: Logical Link Control.* IEEE, 1998.

[45] IEEE, Ed. *802.11-2007 IEEE Standard for Information Technology-Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks-Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.* IEEE, 2007.

[46] IEEE, Ed. *802.3-2008 IEEE Standard for Information technology-Specific requirements–Part 3: Carrier Sense Multiple Access with Collision Detection (CMSA/CD) Access Method and Physical Layer Specifications.* IEEE, 2008.

[47] IEEE, Ed. *802.11n-2009 IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput.* IEEE, 2009.

[48] IEEE, Ed. *802.11s (D4.0) IEEE Draft Standard Local and Metropolitan Area Networks–Specific Requirements–Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications–Amd 10: Mesh Networking.* IEEE, 2009.

[49] IEEE. 802.11 Official Timelines. `http://www.ieee802.org/11/Reports/802.11_Timelines.htm`, 03 2010. Accessed March 28, 2010.

[50] Ingham, D., and Parrington, G. Delayline: a wide-area network emulation tool. *Computing Systems 7*, 3 (1994), 313–332.

[51] International Standards Organization, Ed. *Reference model for Open System Architecture (ISO/TC97/SC16/N227).* International Standards Organization, 1979.

[52] Iperf. `http://iperf.sourceforge.net/`. Accessed June 20, 2010.

[53] ISO – International Organization for Standardization. `http://www.iso.org/iso/home.html`. Accessed April 7, 2010.

[54] Jian, Z., Xiaoyong, L., and Haibing, G. The optimization of Xen network virtualization. In *Proc. International Conference on Computer Science and Software Engineering* (2008), vol. 3, pp. 431–436.

[55] Judd, G., and Steenkiste, P. Repeatable and realistic wireless experimentation through physical emulation. *ACM SIGCOMM Computer Communication Review 34*, 1 (2004), 63–68.

[56] Kaba, J. T., and Raichle, D. R. Testbed on a desktop: strategies and techniques to support multi-hop manet routing protocol development. In *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing* (New York, NY, USA, 2001), ACM, pp. 164–172.

[57] Kamble, N., Nakajima, J., and Mallick, A. Evolution in kernel debugging using hardware virtualization with Xen. In *Proceedings of the 2006 Ottawa Linux Symposium (Ottawa, Canada, July 2006)* (2006).

[58] Karn, P. MACA – a new channel access method for packet radio. In *ARRL/CRRL Amateur radio 9th computer networking conference* (1990), vol. 140.

[59] King, S., Dunlap, G., and Chen, P. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX Annual Technical Conference* (2005), pp. 1–15.

[60] Ko, Y., and Vaidya, N. Location-aided routing (LAR) in mobile ad hoc networks. *Wireless Networks 6*, 4 (2000), 321.

[61] Kolbitsch, C. Atheros wireless device emulation. `http://lists.nongnu.org/archive/html/qemu-devel/2008-02/msg00538.html`, 2 2008. Accessed July 1, 2010.

[62] Krop, T., Bredel, M., Hollick, M., and Steinmetz, R. Jist/mobnet: combined simulation, emulation, and real-world testbed for ad hoc networks. In *WinTECH '07: Proceedings of the second ACM international workshop on Wireless network testbeds, experimental evaluation and characterization* (New York, NY, USA, 2007), ACM, pp. 27–34.

[63] Lacage, M. Direct Code Execution with ns-3. Talk given during the "Workshop on ns-3", March 15th, 2010, Malaga, Spain. `http://www.nsnam.org/workshops/wns3-2010/code-execution.pdf`, 3 2010. Accessed April 9, 2010.

[64] Lacage, M., and Henderson, T. Yet another network simulator. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator* (2006), ACM, p. 12.

[65] LaMarca, A., Chawathe, Y., Consolvo, S., Hightower, J., Smith, I., Scott, J., Sohn, T., Howard, J., Hughes, J., Potter, F., et al. Place lab: Device positioning using radio beacons in the wild. *Pervasive Computing 3468/2005* (2005), 116–133.

[66] LANGLEY, R. NMEA 0183: A GPS receiver interface standard. *GPS world 6*, 7 (1995), 54–57.

[67] Linux Kernel ChangeLog – 2.6.22. `http://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.22`. Accessed May 10, 2010.

[68] LIU, J., YUAN, Y., NICOL, D. M., GRAY, R. S., NEWPORT, C. C., KOTZ, D., AND PERRONE, L. F. Simulation validation using direct execution of wireless ad-hoc routing protocols. In *PADS '04: Proceedings of the eighteenth workshop on Parallel and distributed simulation* (New York, NY, USA, 2004), ACM, pp. 7–16.

[69] LUNDGREN, H., NORDSTRÖ, E., AND TSCHUDIN, C. Coping with communication gray zones in IEEE 802.11b based ad hoc networks. In *WOWMOM '02: Proceedings of the 5th ACM international workshop on Wireless mobile multimedia* (New York, NY, USA, 2002), ACM, pp. 49–55.

[70] mac80211. `http://linuxwireless.org/en/developers/Documentation/mac80211`. Accessed July 1, 2010.

[71] mac80211_hwsim. `http://linuxwireless.org/en/users/Drivers/mac80211\_hwsim`. Accessed July 4, 2010.

[72] MAHADEVAN, P., RODRIGUEZ, A., BECKER, D., AND VAHDAT, A. MobiNet: a scalable emulation infrastructure for ad hoc and wireless networks. In *Papers presented at the 2005 workshop on Wireless traffic measurements and modeling* (2005), USENIX Association, p. 12.

[73] METCALFE, R., AND BOGGS, D. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM 19*, 7 (1976), 395–404.

[74] MITCHELL, C., MUNISHWAR, V. P., SINGH, S., WANG, X., GOPALAN, K., AND ABU-GHAZALEH, N. B. Testbed design and localization in MiNT-2: A miniaturized robotic platform for wireless protocol development and emulation. In *Proc. First International Communication Systems and Networks and Workshops COMSNETS 2009* (2009), pp. 1–10.

[75] MITOLA, J., I. Software radios-survey, critical evaluation and future directions. In *Telesystems Conference, 1992. NTC-92., National* (5 1992), pp. 13/15 –13/23.

[76] NOBLE, B., SATYANARAYANAN, M., NGUYEN, G., AND KATZ, R. Trace-based mobile network emulation. In *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication* (1997), ACM New York, NY, USA, pp. 51–61.

[77] NORDSTROM, E., GUNNINGBERG, P., AND LUNDGREN, H. A testbed and methodology for experimental evaluation of wireless mobile ad hoc networks. In *Proceedings of the Tridentcom* (2005), vol. 2005.

[78] Ns-3: Wifi models. `http://www.nsnam.org/doxygen-release/group___wifi.html`. Accessed May 9, 2010.

[79] Ns-3.9. `http://www.nsnam.org/wiki/index.php/Ns-3.9`. Accessed July 1, 2010.

[80] open80211s. `http://www.open80211s.org/`. Accessed March 28, 2010.

[81] Perkins, C., Royer, E., and Das, S. RFC 3561: Ad hoc On-Demand Distance Vector (AODV) Routing. The Internet Engineering Task Force, 2003.

[82] Postel, J. RFC768: User Datagram Protocol. The Internet Engineering Task Force, 1980.

[83] Pratt, I., Fraser, K., Hand, S., Limpach, C., Warfield, A., Magenheimer, D., Nakajima, J., and Mallick, A. Xen 3.0 and the art of virtualization. In *Proc. of the 2005 Ottawa Linux Symposium* (7 2005).

[84] Pužar, M., and Plagemann, T. NEMAN: A network emulator for mobile ad-hoc networks. Tech. Rep. 321, Department of Informatics, University of Oslo, 3 2005.

[85] Pérennou, T., Conchon, E., Dairaine, L., and Diaz, M. *Two-Stage Wireless Network Emulation.* Springer Boston, 2005, pp. 181–190.

[86] radiotap. `http://wireless.kernel.org/en/developers/Documentation/radiotap`. Accessed April 30, 2010.

[87] Rappaport, T. S. *Wireless Communications: Principles and Practice.* Prentice Hall International, 2002.

[88] Riley, G. The georgia tech network simulator. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research* (2003), ACM, p. 12.

[89] Rizzo, L. Dummynet and forward error correction. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (1998), USENIX Association, p. 31.

[90] Royce, W. Managing the development of large software systems. In *Proceedings of IEEE Wescon* (1970), vol. 26, p. 9.

[91] Sanghani, S., Brown, T., Bhandare, S., and Doshi, S. EWANT: the emulated wireless ad hoc network testbed. In *2003 IEEE Wireless Communications and Networking, 2003. WCNC 2003* (2003), pp. 1844–1849.

[92] Schiller, J. *Mobile Communications*, 2 ed. Addison Wesley, 9 2003.

[93] Seipold, T. Improving emulation of wireless access networks in ns-2. In *ICWMC '07: Proceedings of the Third International Conference on Wireless and Mobile Communications* (Washington, DC, USA, 2007), IEEE Computer Society, p. 41.

[94] Seipold, T. Emulation of radio access networks to facilitate the development of distributed applications. *JOURNAL OF COMMUNICATIONS 3*, 1 (2008), 1.

[95] Shrestha, S., Lee, J., Lee, A., Lee, K., Lee, J., and Chong, S. An open wireless mesh testbed architecture with data collection and software distribution platform. In *Testbeds and Research Infrastructure for the Development of Networks and Communities, 2007. TridentCom 2007. 3rd International Conference on* (may 2007), pp. 1 –10.

[96] Sommerville, I. *Software Engineering*, 8 ed. Addison Wesley, 2007.

[97] Staub, T., Gantenbein, R., and Braun, T. Virtualmesh: an emulation framework for wireless mesh networks in omnet++. In *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques* (ICST, Brussels, Belgium, Belgium, 2009), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1–8.

[98] Swan. `http://web.archive.org/web/20040301090018/http://www.cs.dartmouth.edu/research/SWAN/`, 2004. Accessed June 21, 2010.

[99] Tanenbaum, A. S. *Computer Networks*, 4 ed. Pearson Education, 2003.

[100] Tcpdump/libpcap. `http://www.tcpdump.org/`. Accessed July 1, 2010.

[101] The freebsd project. `http://www.freebsd.org/`. Accessed June 27, 2010.

[102] The Linux Kernel Archives. `http://kernel.org/`. Accessed Apr 18, 2010.

[103] The ns-3 network simulator. `http://www.nsnam.org/`. Accessed April 21, 2010.

[104] The Temporary Netperf Homepage. `http://www.netperf.org/netperf/`. Accessed June 15, 2010.

[105] Thiem, L., Riemer, B., Witzke, M., and Luckenbach, T. Rfid-based localization in heterogeneous mesh networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems* (New York, NY, USA, 2008), ACM, pp. 415–416.

[106] Tourrilhes, J. Wireless tools for linux. `http://www.hpl.hp.com/personal/Jean\_Tourrilhes/Linux/Tools.html`. Accessed July 1, 2010.

[107] Varga, A., et al. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)* (2001), pp. 319–324.

[108] Varga, A., and Hornig, R. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops* (2008), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), p. 60.

[109] Walton, S. *Linux Socket Programming.* Sams, Indianapolis, IN, USA, 2001.

[110] Wang, S., and Lin, C. NCTUns 5.0: A network simulator for IEEE 802.11 (p) and 1609 wireless vehicular network researches. In *IEEE Vehicular Technology Conference (VTC-Fall)* (2008), pp. 1–2.

[111] Wehrle, K., Pählke, F., Ritter, H., Müller, D., and Bechler, M. *Linux Networking Architecture – Design and Implementation of Networking Protocols in the Linux Kernel.* Prentice-Hall, 5 2004.

[112] Weingärtner, E., Schmidt, F., Heer, T., and Wehrle, K. Synchronized network emulation: Matching prototypes with complex simulations. In *Proceedings of the First Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics'08), co-located with ACM SIGMETRICS 2008* (Annapolis, MD, USA, 2008).

[113] Weingartner, E., vom Lehn, H., and Wehrle, K. A performance comparison of recent network simulators. In *IEEE International Conference on Communications, 2009. ICC'09* (2009), pp. 1–5.

[114] Wi-Fi Alliance. `http://www.wi-fi.org/`. Accessed March 19, 2010.

[115] Wireless Tools for Linux. `http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html\#wext`. Accessed May 10, 2010.

[116] Wireshark. `http://www.wireshark.org/`. Accessed June 15, 2010.

[117] Wu, G., Singh, S., and Chiueh, T.-c. Implementation of dynamic channel switching on ieee 802.11-based wireless mesh networks. In *WICON '08: Proceedings of the 4th Annual International Conference on Wireless Internet* (ICST, Brussels, Belgium, Belgium, 2008), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1–9.

[118] Xen homepage. `http://www.xen.org/`. Accessed April 20, 2010.

[119] Yokota, H., Idoue, A., Hasegawa, T., and Kato, T. Link layer assisted mobile IP fast handoff method over wireless LAN networks. In *Proceedings of the 8th annual international conference on Mobile computing and networking* (2002), ACM, p. 139.

[120] Zhang, Y., and Li, W. An integrated environment for testing mobile adhoc networks. In *MobiHoc '02: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing* (New York, NY, USA, 2002), ACM, pp. 104–111.

A

Appendix

As Chapter 4 describes the implementation only on a conceptual level, it has not yet been covered how to actually use it in practice. The following Appendix A.1 explains how to set up the network simulation and wireless network card drivers. Appendix A.2 supplements Chapter 5 with additional figures.

A.1 Usage Howto

Each wireless network emulation setup consists of a network simulation and at least one system hosting an emulated wireless network card driver. These two components can in principle be run in the same Linux system, but the network card driver will typically be used in a separate system (real or virtual machine). As the network card driver registers upon initialization at the simulation, the network simulation has to be started first.

Simulation setup

The following code snippet shows the necessary steps which turn a regular ns-3 Wi-Fi simulation into a setup supporting wireless network emulation:

```
GlobalValue::Bind ("SimulatorImplementationType",
                   StringValue ("ns3::RealtimeSimulatorImpl"));
GlobalValue::Bind ("ChecksumEnabled",
                   BooleanValue (true));
// Here goes rest of simulation setup ...
WifiEmuBridgeHelper wbridge;
wbridge.SetAttribute("ClientId", IntegerValue(42));
wbridge.Install(c.Get(0), staDevice.Get(0));
```

Name	Type	Default Value
`WifiEmuBridge::ClientId`	uint16	1
`WifiEmuCommType`	Enum	`CommTypeUdp`
`WifiEmuCommUdpReceiveAddress`	`Ipv4Address`	0.0.0.0
`WifiEmuCommUdpReceivePort`	uint16	1984

Table A.1 Configuration options for the wireless emulation bridge classes.

First of all, two general settings are necessary in order to support the communication with real devices. The simulator type has to be set to real-time (line 1) and the checksum calculation has to be enabled (line 3).

After that, the wireless emulation bridge has to be installed on all nodes which shall serve as gateway nodes for real implementations (lines 6 to 8). These nodes are regular Wi-Fi nodes containing a `WifiNetDevice` whose MAC type has been either set to station or adhoc. As the gateway nodes only serve as a placeholder for real systems, there is no Internet stack installed on them.

The class `WifiEmuBridge` and its helper classes can be configured through a few options which are listed in Table A.1. The most important option is the `ClientId` attribute of `WifiEmuBridge`, which is used to match the corresponding gateway nodes to driver instances registering at the simulation. Using `WifiEmuCommType` one could select a different communication method than UDP, but there is none implemented at the moment. Finally, the two settings `WifiEmuCommTypeUdpReceiveAddress` and `WifiEmuCommUdpReceivePort` determine the UDP endpoint on which the simulation listens for messages of the drivers. In order to express that there is only one instance of `WifiEmuComm`, the latter three options have been implemented as a GlobalValue instead of an object-specific attribute.

Using the steps and configuration options described above, the resulting network simulation can be executed like any regular ns-3 simulation. When the simulation is set up and running, the Wi-Fi emulation bridges installed on the gateway nodes wait for network card drivers to register.

Driver setup

After the simulation is set up and running, instances of the wireless network card driver can be loaded into the Linux systems which shall be used to execute the software under test. As described in Chapter 4, the driver is implemented as two separate kernel modules: `wifi-emu` for the generic functionality and `wifi-emu-udp` for the UDP-specific part.

These two modules are mutually dependent on each other, but are built in a fashion that `wifi-emu-udp` makes use of the functionality exported by `wifi-emu`, for which reason `wifi-emu` has to be loaded first. When using the insmod command, the two modules have to be loaded one by one. The modprobe command can detect this dependency, so that it suffices to issue the command for `wifi-emu-udp`.

During initialization of the modules, the driver tries to register at the network simulation. If it succeeds, a wireless network device called `wemu0` is installed in the system. From this point on, the network emulation is set up and the wireless network device can be used like a regular 802.11 Wi-Fi card. If the registration fails

Module	Name	Type	Default Value
`wifi-emu`	`client_id`	ushort	1
`wifi-emu`	`buffer_size`	ulong	65536
`wifi-emu`	`enable_buffer`	bool	1
`wifi-emu-udp`	`peer_addr`	string	”127.0.0.1”
`wifi-emu-udp`	`peer_port`	ushort	1984

Table A.2 Configuration options for the wireless network card driver.

or no network connection to the simulation can be established, the issued command (insmod or modprobe) returns an error. The created network device exists in the system until the module `wifi-emu-udp` is unloaded later-on.

The configuration of the driver takes place through a number of module parameters which are listed in Table A.2. The identifier configured through `client_id` is included in all messages sent to the simulation and is used to match the corresponding gateway node. By using `buffer_size` one can configure the size of the virtual transmit buffer or disable this functionality completely using `enable_buffer`. The two parameters `peer_addr` and `peer_port` determine the UDP endpoint of the network simulation to connect to.

A.2 Additional Figures

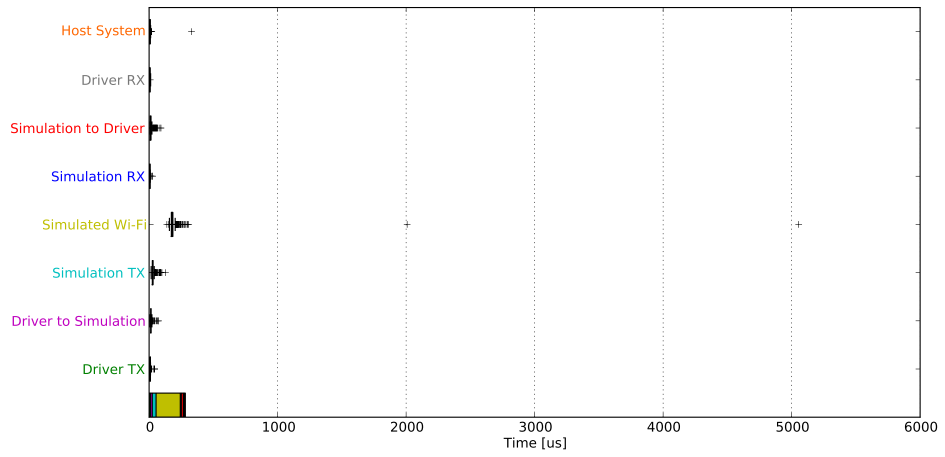


Figure A.1 Figure 5.8 with outliers.

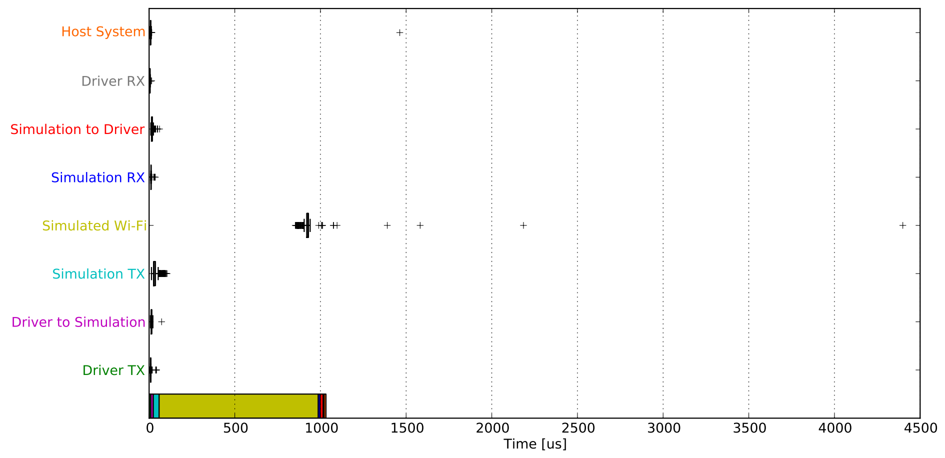


Figure A.2 Figure 5.9 with outliers.