Studiengang: Informatik

Prüfer: Prof. Dr. rer. nat. Dr. h. c. K. Rothermel

Betreuer: Dipl. Inf. Susanne Bürklen
Dipl. Inf. Dominique Dudkowski
Dipl. Inf. Jörg Hähner
Dr. rer. nat. Pedro José Marrón

CR-Klassifikation: C.2.1, C.2.2, C.4, I.6.1, I.6.8

Diplomarbeit Nr. 2195

Entwurf eines adaptiven und interaktiven Systems für die Analyse von Algorithmen zur Datenverwaltung in mobilen Ad-hoc-Netzen

Michael Moltenbrey

begonnen am: 22. März 2004

beendet am: 21. September 2004

Fakultät für Informatik, Elektrotechnik und Informationstechnik
Institut für Parallele und Verteilte Systeme
Abteilung Verteilte Systeme
Universität Stuttgart
70569 Stuttgart

Abstract

With the increasing importance of mobile and wireless applications in our daily life, methods of performance analysis of these applications becomes more and more important.

However, mobile and wireless networks implicate new challenges. Among these challenges are mobility of network nodes, limited resources of mobile devices and the unreliability in communication. Mobility causes frequent changes in the network topology which have to be compensated.

A common performance analysis method is network simulations. Numerous network simulators have been developed over the years. Traditional network simulators focus on wired networks cannot be applied to mobile and wireless networks since they are not able to cope with these challenges. However, an increasing number of simulators have been adapted to fit the needs of mobile and wireless networks or have been newly developed. Only few of them are able to deal with mobile ad hoc networks.

Additionally, state-of-the-art network simulators have a high resource consumption and lack certain features, like modularity, adaptability and interactivity that would ease the use of simulations. Therefore, we develop the central components of a new network simulation tool which supports these features.

We provide an implementation of a prototype system which supports our design concepts and which allows the simulation of wireless scenarios.

Contents

List of Figures

List of Tables

Listings

1. Introduction

1.1. Motivation

Mobile and wireless networks have gained importance in recent years. Last but not least this traces back to the boom of mobile phones, the price decline of hardware used for the setup of wireless networks, like wireless network cards or access points and the desire to have access to information every where at every time. However, those new technologies implicate new challenges, ranging from resource limitations of mobile devices, like limited memory and energy, to complex and continuously modifying network topologies, e.g. due to node movement.

In the beginning, traditional techniques and protocols for fixed wired networks were applied to wireless technology. However, it soon became obvious that mobile and wireless networks bring up new problems that cannot be solved by using the traditional techniques. Mobile networks require new techniques and numerous adaptations.

In wired networks, network nodes, like PCs, typically remain fixed at one position. Therefore, the topology of the network changes only infrequently. However, in mobile and wireless networks each unit may often move and thereby causes frequent changes in the topology. The protocols used in an environment like this have to continuously adapt to these varying situations.

In addition, communication via wireless links is much more prone to interference and therefore more unreliable than in wired networks. Protocols have to take that into account as well. Both topics are typically not addressed by protocols used in wired networks to the degree required in wireless ones.

This influences most parts of the protocol stack. For example, the MAC layer needs to be adapted, because the Ethernet MAC layer is not sufficient for wireless networks. It provides carrier sense for multiple access with collision detection which does not solve all the problems in wireless networks which are still exposed to several problems like the hidden terminal problem and the exposed terminal problem.

Going up the protocol stack, the network layer also needs adaptations. New routing protocols have to be developed that are able to cope with the mobility of nodes.

In order to support the development of new protocols for wired networks, Network simulators had been implemented. With the increasing importance of wireless networks, simulators for these networks are needed. Some existing simulators for wired networks, like NS-2, have been adapted to fit the needs of wireless networks and some new simulators have been specially designed for wireless and mobile networks, like GTNetS and GloMoSim. Most of them focus on infra-structure based wireless networks and lack sufficient support of mobile ad hoc networks. Additionally, their flexibility and adaptivity is limited, e.g. there is no possibility to change simulation scenarios during the runtime of the simulation, and resource consumption of these network simulators is typically quite high.

It is necessary to develop tools tailored to the special needs of mobile ad hoc networks. A tool like this supports the development of new protocols for mobile and wireless networks and thereby mobile ad hoc networks, too.

1.2. Thesis Layout

This thesis covers the design and development of the central parts of a new network simulator and demonstrator based on discrete event simulation. Its primary focus is on mobile ad hoc networks. However, wired networks are supported, too.

In Chapter 2 we will give some basics on mobile ad hoc networks in Chapter 2. We will provide all information necessary to understand the design decisions made in this thesis.

The discussion and analysis of the related work will take place in Chapter 3. There we will examine the different types of analysis methods, ranging from real-world tests to simulations. Afterwards, we will glance at different simulators that can be used for the simulation of mobile and wireless networks.

In Chapter 4 we will provide an overview of the simulator's complete system architecture. We will specially address those parts relevant for this thesis. This chapter forms the basis of our design approaches of our simulator.

Ensuing, Chapters 5 and 6 provide the design of our approach in detail. Whereas Chapter 5 designs the central components of our simulator. Chapter 6 describes the design of the simulation components that are required to perform a simulation. The implementation aspects will be pointed out in Chapter 7.

After discussing our approach we will evaluate our simulator by using a simple scenario. The results are presented and discussed in Chapter 8.

Finally, we conclude the thesis in Chapter 9 and present future work in Chapter 10.

2. Principles of Mobile Ad Hoc Networks

This chapter provides an overview of mobile and wireless networks. It introduces concepts necessary for understanding the Cubus design concepts provided in the following chapters.

First, we give an overview on what types of computer networks exist and how they can be classified. Secondly, we describe the specialties of *mobile ad hoc networks*. Their advantages and problems are pointed out. Thirdly, wireless networks require some changes in the standard layer model. These will be outlined in Section 2.3. This is necessary to understand some basic design issues of our simulator Cubus. Fourthly, a closer look is taken at a specific wireless LAN implementation, namely IEEE 802.11, because this implementation was ported to Cubus in the course of this thesis.

2.1. Classification of Networks

Computer networks can be categorized into several groups, as depicted in Figure 2.1. The most obvious classification is between *wired networks* and *wireless networks* [Schi00]. Wireless networks can be further subdivided into two groups: in *structured networks* which

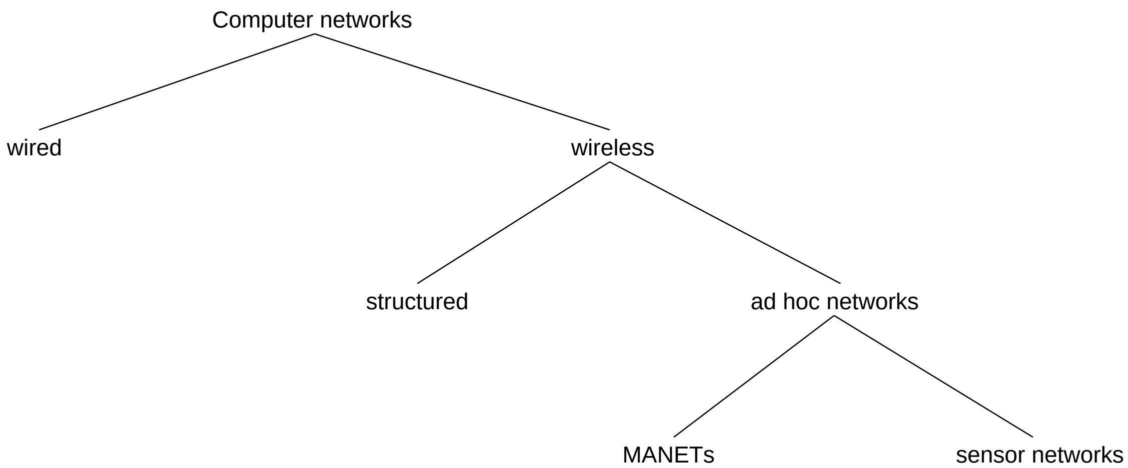

Figure 2.1.: Classification of computer networks

use specific infrastructure hardware (see Section 2.1.2) and *ad hoc networks* [Perk01]. The latter do not use any kind of special infrastructure hardware (see Section 2.1.3). Important classes of Ad hoc networks are *sensor networks* [AWSC02] and *mobile ad hoc networks* (MANETs) [Bake02].

2.1.1. Wired Networks

Wired networks are the first kind of network used to connect computers. Therefore, they have a very long tradition and computer scientists, engineers and programmers have developed many protocols and network components over the last decades.

Their main characteristics are fixed, wired connections between the different components. Nodes in wired networks use common transmission protocols like IEEE 802.3 and Token-Ring. Two different types of connections exist, “busses” and “point-to-point links”. In a bus system, all network nodes are connected to the same communication channel. While in point-to-point networks connections between each pair of nodes exist. The latter requires routing, since several nodes may be on the path from the sender to the receiver. Communication in bus systems takes place via broadcasts. The sender broadcasts its message on the bus and the appropriate receiver takes it from there.

The components of a wired network are usually stationary. Additionally, “plugin” stations for nomadic devices are available. At these plugin stations, mobile devices can be attached to a wired network. Due to their nature, wired networks are primarily used in areas, where the network topology does not change very often, i.e. where the infrastructure is kept as is for a longer period of time.

2.1.2. Wireless Networks

Wireless networks are much more flexible and dynamic than wired ones. They can be subdivided into networks which require an infrastructure and networks which lack this infrastructure. Figure 2.2 shows an infrastructure-based wireless network (based on a figure from [Schi00]).

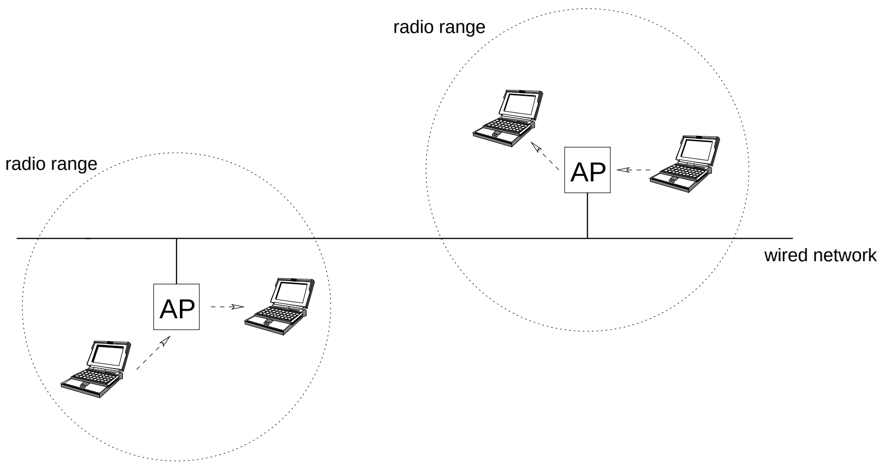

Figure 2.2.: Wireless, infrastructure-based network

This kind of network requires special hardware, e.g. *access points* (AP) which are usually attached to a conventional wired network. All communication takes place via these access points, i.e. no two nodes can communicate directly with each other.

The attachment of access points to a wired network allows communication between different mobile networks, even though they might not be in radio range of each other. Communication between two mobile nodes of different mobile networks happens via the wired network to which they are attached by using the access points. As shown in Figure 2.2, a node of the network on the lower left side might be able to communicate with a node in the upper right network.

This simplifies the network’s structure, since most network functionalities, like routing mechanisms, are encapsulated in the access points. These access points have full control of medium access, which leads to the avoidance of collisions. Within one mobile network the access points are the only “network node” that controls access to the medium. Since access points manage most things of the network, the mobile nodes themselves can be kept simple. Infrastructure-based wireless networks are easy to manage but are not as flexible and dynamically adaptable than ad hoc networks.

Today, this kind of network is quite common, e.g. cellular phone networks.

2.1.3. Ad Hoc Networks

Ad hoc networks do not require an infra-structure network. Rather, mobile nodes communicate directly with each other. If two nodes are not within radio range of each other, they can use the forwarding functionality of another node to establish a connection, i.e. the message travels from one node to another until it reaches its destination.

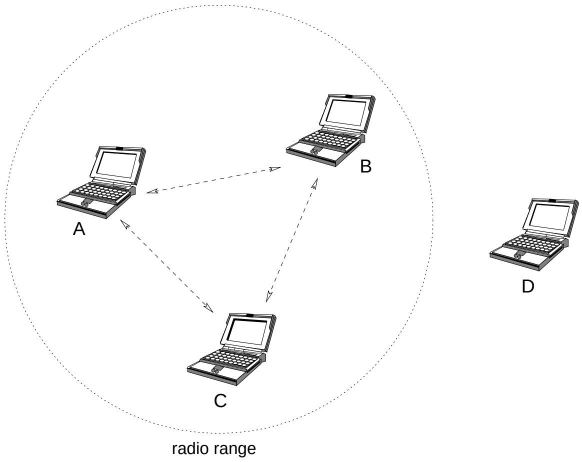

Figure 2.3.: A simple ad hoc network

Figure 2.3 shows a simple ad hoc network. Nodes A, B and C are within radio range of each other and can communicate directly via a single-hop. But if another node, e.g. D, is out of range, there is no way to communicate with that node. This shows a disadvantage of ad hoc networks: Communication between different ad hoc networks can hardly be realized.

Another disadvantage is complexity. All nodes need to implement at least simple medium access mechanisms and need to detect collisions themselves. Therefore, nodes of ad hoc networks are much more complex than those of infrastructure based networks.

However, ad hoc networks are easy to manage and establish. Since they do not require an infrastructure network, they are much more flexible and their use is possible in a broader range of scenarios, e.g. for disaster relief.

Depending on the frequency of structural changes in the network, ad hoc networks can be subdivided into *mobile ad hoc networks*, or *MANETs*, and *sensor networks*.

Sensor Networks

Basically, sensor networks, as described in [AWSC02], consist of a large number of sensor nodes. Sensor nodes are very small units, equipped with components for sensing, data processing and communication. A commonly used sensor node type is the so called *SmartIt* [TecO04]. Since sensor nodes only have a small amount of memory, short distance communication is a common way of communication regarding sensor networks.

A single node's position is irrelevant for the functionality of the network. In most cases, after a node gains its initial position, this position is considered fixed. Nodes can be randomly scattered over the area of observation. As soon as the network starts working, routing tables are created, which include information that is used to determine the nodes to which data can be sent. These tables are usually only altered after node failures have been discovered.

Sensor networks are easy to set up, since they rely heavily on self-organization which imposes difficulties on algorithm design. Currently, their main fields of use can be found in disaster relief and in the military sector.

Mobile Ad Hoc Networks

Mobile ad hoc networks are far more complex to realize than infrastructure-based networks or wired networks. They consist of independent mobile nodes that organize themselves. Other than in sensor networks, their structure and topology may change rapidly. A definition of MANETs is given by the *Internet Engineering Task Force* in [Bake02]:

> A "mobile ad hoc network" (MANET) is an autonomous system of mobile routers (and associated hosts) connected by wireless links – the union of which form an arbitrary graph. The routers are free to move randomly and organize themselves arbitrarily; thus, the network's wireless topology may change rapidly and unpredictably. Such a network may operate in a standalone fashion, or may be connected to the larger Internet.

In other words, MANETs are a set of wireless mobile nodes, which are able to establish a temporary network among themselves without using any kind of infrastructure and without central management.

2.2. Challenges of Wireless Communication

Until recently, computer networks were based on wired links. However, in the last years wireless communication has become more and more important. With wireless communication, new challenges arise.

In wires, signals, the physical representation of data, spread in defined directions along the wire. The cable guides the signal to its destination. This spreading of signals called *signal propagation*. Propagation in a cable happens in a determined way. Every cable has specific characteristics which hold as long as it isn't damaged. Therefore, we know how the signal behaves at every point in the cable and every point of time.

Unfortunately, that does not hold for wireless communication. Considering signal propagation in a wireless medium, e.g. air, we have to take *interference* into account. This phenomenon is discussed in the following sections in greater detail.

Additionally, different types of dissemination are available. These are based on different types of antennas. Another major difference between MANETs and wired networks is the possible high mobility of MANET nodes. This aggravates reception and propagation of signals. Moreover, mobile nodes are potentially more unreliable than fixed nodes in wired networks, e.g. due to lack of energy.

2.2.1. Frequencies Used in Wireless Communication

Every communication in wireless networks can take place in a great variety of frequency bands, ranging from *low frequency* (LF) bands up to *super high frequency* (SHF) bands and even extremely high frequency *EHF* bands. For more detailed information, see [Schi00, Tane03]. Table 2.1 shows some of the most prominent representatives of wireless technologies.

Technology	Bluetooth	GSM	UMTS	Wireless LAN
Frequency (MHz)	2402 - 2480	890 - 915 935 - 960 1710 - 1785 1805 - 1880	1885 - 2025 2110 - 2200	2400 - 2483

Table 2.1.: Overview of different technologies

2.2.2. Signal Propagation

This section gives a glimpse on the basics of signal propagation. This is necessary to understand the signal's behavior in a medium, like air.

Signals are the physical representation of data. They are formed by electromagnetic waves. Several frequency bands are used for wireless communication, as discussed in Section 2.2.1. There is a correlation between the electromagnetic wavelength λ and the frequency f:

$$\lambda = \frac{c}{f}$$

where c is the speed of light.

Signals propagate differently in wireless media than in wired ones. The are no longer guided by a cable and can therefore spread in multiple directions depending on antenna and environment. How signals propagate partially depends on the antennas used. We will only consider *istrotropic antennas* in free space with our simulator, which send out signals equally in all directions. More detailed and realistic antennas are part of future work.

Using isotropic antennas results in a signal propagation resembling a sphere. Every signal is emitted with a specific energy. Let us assume that a sender emits signals with power P_t. Since the signal spreads spherical, it covers an area of $4\pi d^2$ when it reaches a distance d from the sender. The power at a specific point of this area (*power flux density*) S is given by

$$S = \frac{P_t}{4\pi d^2}$$

While a signal propagates, there is a transmission loss. This loss is given by the interaction of receiver and sender, or more exactly between their antennas. Let us assume that the receiver's antenna has an effective aperture A_r. Then the power P_r with which it receives a signal is determined by

$$P_r = S \cdot A_r = P_t \cdot \left(\frac{\lambda}{4\pi d}\right)^2$$

Assuming the signal propagates in a ideal medium, where the signal does not weaken due to the influence of other phenomena, there still is a specific loss of energy on the path from sender to receiver, the so called *free space path loss* L_p.

$$L_p = \frac{P_t}{P_r} = \left(\frac{4\pi}{c}\right)^2 \cdot f^2 \cdot d^2$$

Having seen the path loss of a signal during its propagation, we can understand what other problems may arise. Assuming spherical propagation, we can distinguish at least three shells in different distances to the sender. The first shell is called the *transmission shell*. This shell is a sphere with radius r_t around the sender. Within this range the signal is strong enough to be properly received and processed by other nodes.

As we increase the distance to the sender, the signal gets weaker due to path loss. It converges to the background noise. This is where the second shell, the *detection shell*, comes into account. It ranges from distance r_t to r_d from the sender. Within this shell, nodes can detect that there is a signal, but the error rate is too high to process signals correctly.

The third shell, the *interference shell*, is where the signal is indistinguishable from noise. No information can be taken from the signal anymore. It interferes only with other signals. In the worst case, interference can even lead to wave extinction. See [Orfa04] for more and additional details on interference and influences on signal propagation, like *dispersion*, *reflection* and *shadowing*.

Schiller [Schi00] describes *multipath propagation* (MP) which might be of interest. MP happens in almost all real life scenarios. There is no single path via which signals are transmitted to a receiver. In fact, there are infinitely many paths. Since not all paths have the same length, the receiver gets the signals temporally delayed. This makes it more difficult to get a proper signal. A single, clear “peak” from the sender is transformed to something smeared at the receiver’s side. Since this descends much to deep into electronical and physical domains and leaves the scope of this thesis, the reader is referred to the literature for additional information, e.g. [Schi00], [Orfa04] and [Tane03].

2.2.3. Collisions

Collisions may happen in all kind of networks, whatever technology (wired, wireless) is used. They occur if two senders try to access and use the same medium at the same time. Thereby, the frames sent collide and both transmissions fail. Consider two senders attached to a single wire cable. Sender A wants to send data to B and vice versa. If B sends its data on the cable while A is already sending, a collision happens.

Collisions turn out to be a serious problem in communication and therefore need to be avoided or at least detected. We will describe the basic principles in wired and wireless networks in the following sections.

Collision Detection in Wired Networks

Since collisions are an old, well known problem, multiple solutions exist. Among them are *ALOHA* [Abra85], *slotted-ALOHA* [Robe72] and *Carrier Sense Multiple Access with Collision Detection* [IEEE02]. The latter has become part of the Ethernet (IEEE 802.3) standard.

The basic principle of CSMA/CD is easy to understand: A node senses the medium via which it wants to send. If the medium is free, i.e. no other node is sending data on this medium, it sends its data. If the medium is busy, i.e. another node already occupies this medium, the sensing node waits for a random amount of time. After that, it tries again.

If a node realizes that another node has sent data after it has sent his data, he cancels his transmission, since a collision happens and the sent frames are going to be faulty. This situation can occur if two nodes, waiting for a medium getting free and after it was freed, decide to send their data at the same time. A detailed description of CSMA/CD can be found in [Tane03].

Collision Detection in Wireless Networks

At first, it may be tempting to use CSMA/CD in wireless networks, since it is an easy to use and already well-tested method. But it turns out that it will not work with wireless networks.

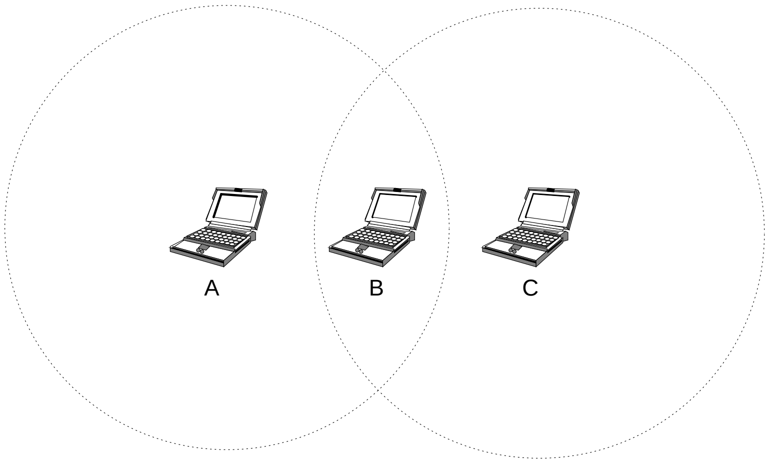

Figure 2.4.: Hidden terminal problem

Consider the scenario shown in Figure 2.4. There are three nodes A, B and C. For simplicity, we only consider the transmission range of these nodes and not the detection and interference range. Assume that A is in transmission range of B, but not of C and that C can reach B, but not A. B can reach A and C. Node A does not know anything about the transmissions of C since A is outside of C's detection range and vice versa.

The problems arising when using CSMA/CD can be realized looking at the following scenarios.

Nodes A and C want to send something to B. Node A senses the medium and finds out that it is free. Therefore, it starts sending its data to B. Now C senses the medium, and since A is out of range, it assumes that the medium is free. Hence, C starts sending to B. But this causes a collision at B. A and C continue transmitting without knowing that a collision occurred. This is called the *hidden terminal problem* [KlTo75].

Other problems exist when using CSMA/CD, like the *exposed terminal problem.* Further information can be found in [Tane03], p. 301, and [Schi00], p. 62.

To solve these problems, new approaches are required. One of the first approaches was the development of MACA [Karn90]. MACA introduces the concepts of *Request To Send* (RTS) and *Clear To Send* (CTS) frames. With MACA, a node sends a small frame (RTS) of typically 30 bytes to the receiver. This frame contains information about the actual data frame's length. If the receiver is ready to receive, it acknowledges with a CTS frame. The CTS also contains information about the length. Any other node in transmission range of the sender must not send anything until CTS is received by the sender. Nodes in transmission range have to wait until the actual data frame is completely received. The time can be determined by using the length included in the CTS frame.

Another approach is MACAW [BDSZ94], which is an optimization of MACA. The most prominent representative is *Carrier Sense Multiple Access with Collision Avoidance* (CS-

MA/CA) [IEEE99a] which is part of the IEEE 802.11 standard and is discussed in Section 2.3.

2.3. An Adapted Layer Model for Mobile Networks

The previous sections described some of the basic characteristics of wireless communication and its differences to wired networks. These characteristics require adaptations in the standard ISO/OSI reference model. The changes are outlined in this section. Primary focus is put on the physical and MAC layer. Both are necessary to understand some basic principles of the Cubus design. The primary focus of this consideration is on IEEE 802.11 [IEEE99a], the most wide-spread realization of wireless networks.

As determined by its name, the IEEE 802.11 standard belongs to the IEEE 802.x family. Therefore, there are many things in common. Basically, IEEE 802.11 describes the *physical layer* as well as the *MAC layer*. Figure 2.5 shows the basic layout of the IEEE 802.11 protocol stack as defined by the 1999 standard.

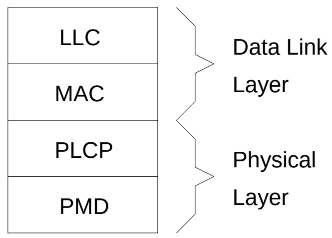

Figure 2.5.: IEEE 802.11 protocol stack

Only the relevant parts of the stack are shown. The layers above *data link layer* are common to all IEEE 802.x, which are based on the ISO/OSI model.

The *physical layer* splits into two sublayers, the *physical layer convergence protocol* (PLCP) and the *physical medium dependent* (PMD). Both of them are described in Section 2.3.1.

The *data link layer* is also broken down into two parts. The sublayer above the physical layer is the *medium access control* (MAC) layer which controls medium access. The next layer is the *logical link layer* (LLC) which is common to all 802.x standards. It hides the differences of the MAC layers to the upper layers. The data link layer, especially the MAC layer, is described in more detail in Section 2.3.2.

The description in the following subsections is based on [Tane03], [Schi00] and [IEEE99a].

2.3.1. IEEE 802.11 Physical Layer

The 1997 standard provides three basic transmission technologies for IEEE 802.11. The first one is based on *infrared*. This is not discussed here, since its relevance for this thesis is low. The other two technologies are based on radio transmission. They are *Frequency Hopping Spread Spectrum* (FHSS) and *Direct Sequence Spread Spectrum* (DSSS). Both of these radio transmission technologies work in the 2.4 GHz ISM band (see Section 2.2.1). Their throughput is limited to 1 or 2 Mbit/sec.

In 1999, a revision of the standard [IEEE99b] added two new technologies. They form the foundation of wireless high-speed LANs by providing rates of 54 Mbit/sec or 11 Mbit/sec. Namely these are *Orthogonal Frequency Division Multiplexing* (OFDM) and *High Rate Direct Sequence Spread Spectrum* (HR-DSSS).

The IEEE 802.11 physical layer comprises two sublayers, PLCP and PMD. PLCP comprises user data fragmentation, media access and encryption. Additionally, it provides a common service access point which is independent of the transmission technology and a carrier sense signal, *clear channel assessment* (CCA).

The PMD handles encryption and decryption of signals. It also handles *modulation*, i.e. the process of translating the base band signal of a transmitter, an analog signal, to the carrier signal.

Modulation is required if digital data is going to be transmitted via a medium, that is not capable of handling digital signals. Wireless media require analog signals. Therefore, the digital data is translated to the analog base band signal.

2.3.2. IEEE 802.11 MAC Layer

Among other things, the IEEE 802.11 MAC layer [IEEE99a] controls access to the medium. The standard defines three basic access methods.

1. *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA)
2. *CSMA/CA with RTS/CTS*
3. *Contention-free polling method*

The first method is defined as mandatory. Additionally, all implementations of IEEE 802.11 must react properly to RTS and CTS packets, even though they may not realize the second access method.

CSMA/CA and CSMA/CA with RTS/CTS can be classified as *Distributed Coordination Function* (DCF), whereas the third one is sometimes called *Point Coordination Function* (PCF). The MAC mechanisms themselves can be characterized as *Distributed Foundation Wireless Medium Access Control* (DFWMAC).

In addition to these three access methods, some time intervals are defined which control waiting and access. These times are called *DCF inter-frame spacing* (DIFS), *Short inter-frame spacing* (SIFS) and *PCF inter-frame spacing* (PIFS). Further information can be found in the Standard itself [IEEE99a], p. 74 or in the literature [Tane03, Schi00].

We will take a closer look at the first two access methods which are realized for Cubus.

CSMA/CA

This is the only basic access method defined by the standard which is mandatory. Its functionality is shown below.

A node wishing to send a frame after it received a request of an upper layer algorithm, senses the medium to detect whether it is idle. Thereto, it uses the CCA provided by the physical layer. If it finds the medium busy, the node has to wait for DIFS and an additional *Random Backoff Time*. This backoff time is decremented as long as the medium is idle. The problem with this approach is its potential unfairness. A node may wait for an arbitrary amount of time and other nodes which requested transmission after that, one may have already finished.

As a refinement to this approach, a *Backoff procedure* was introduced. This backoff procedure is invoked when the medium is found to be busy, but a node wants to transmit a frame. Then, the *Backoff Timer* is set to a random backoff time. It is checked if there is any activity during each backoff slot. The time is only decremented if there is no activity, i.e. the medium is idle.

If the medium is busy during a slot, the timer is stopped and the node has to wait at least a DIFS period. After that, the timer is resumed. The transmission takes place as soon as the timer reaches zero. The main advantage of this approach is that it is fair. Nodes that have been waiting for a longer periods of time are favored, since they have smaller values for their backoff times.

The principle described above works fine with broadcast frames. Regarding Unicast, special *ACK* frames for acknowledgment are required. Acknowledgment ensures the correct reception of the data frames. If no ACK is received by the sender, it assumes that something went wrong and retransmits the frames.

Unfortunately, this method does not prevent the hidden terminal problem (see Section 2.2.3). This is where the second access method comes into play.

CSMA/CA with RTS/CTS

This access method is based on MACA which was introduced in Section 2.2.3 and solves the hidden terminal problem.

Besides the two special control packets RTS and CTS which extend the previous version, the standard specifies the *net allocation vector* (NAV) which gives the next point of time when the node may try to access the medium again. It is a kind of virtual carrier-sense mechanism. It tries to model future traffic by using duration information provided by the RTS and CTS packets.

The basic procedure is very similar to MACA. The sender first transmits an RTS packet. It includes the receiver and the duration of the data transmission. Every node that receives this RTS packet, sets its NAV according to the data transmission's duration.

The receiver of the data transmission waits for a SIFS period after it has received the RTS packet. After this SIFS period, it replies with a CTS packet. This packet contains the duration of the data transmission again.

All nodes receiving the RTS or the CTS need to adjust their NAVs, i.e. medium access for these nodes is delayed.

2.4. Summary

In this chapter, we have discussed the different types of networks and their characteristics, ranging from wired to wireless networks. Infrastructure-based networks and ad hoc networks are wireless networks which impose several new challenges, e.g. signal propagation is more complex than in the wired case. In general, signals propagate different in wireless networks, e.g. according to free space path loss. When signals propagate, they can interfere and cause collisions.

We have discussed an adapted layer model, provided by IEEE 802.11, which solves most of the problems imposed by wireless networks. We have seen that approaches of wired networks are not sufficient. For example, the hidden terminal problem is solved by employing CSMA/CA with RTS and CTS frames but cannot be solved using the IEEE 802.3 CSMA/CD approach.

3. Related Work

In this chapter, we present related work in the field of network simulation and discuss it with regard to our demands. The first part of this chapter covers methods and aspects of performance and behavioral analysis of algorithms in computer networks. The second part deals with a selection of network simulators. We discuss these simulators related to our needs.

3.1. Performance Evaluation Methods

Years ago, in the 1970s and 1980s, computer networks had been rarely used. However, by the advent of the Internet, computer networks have become more and more important. The problems that have arisen by the use of networks, have enforced the development and design of new protocols and algorithms.

These new protocols and algorithms need to be properly tested and evaluated to get reliable information about their behavior. Basically, there are three ways for testing and evaluating.

The first way is comprised by the use of *real-world network environments*, by employing appropriate testbeds, or in short *real system testbeds*. Another way is *emulation*, a relatively new research topic. These two approaches are not subject to this thesis. Therefore, we only discuss them briefly.

The third approach and the main focus of this thesis is *simulation*, a wide-spread and established test method. We closely discuss the different types of simulation in this chapter.

3.1.1. Real System Testbeds

Testing an application in a real-world environment using “authentic” hardware is the most obvious idea when thinking about possible ways of testing, because real hardware is used under real-world conditions. In these scenarios, actual mobile devices, like notebooks or cell phones, are employed. They are the mobile nodes of the scenario. This prevents adulterations of the results, since mobile nodes are not simulated. Simulation always leads to certain inaccuracies.

Additionally, the actual protocol implementations are used in the same way, they are going to be used in their application at a later date. This prevents adulterations, too. No abstraction has to be applied, as it would be required in simulations or emulations.

We examine tests in real systems by considering a similar approach outlined in [MaBJ99] and [MaBJ00], as an example. The goal of this approach is the development of a physical testbed that allows investigation of protocols for mobile ad hoc networks dealing with real-world conditions.

In this project, multiple mobile nodes (cars) are employed. Each of these nodes is equipped with a WaveLAN [Luce04] capable network interface and a GPS receiver. In addition to

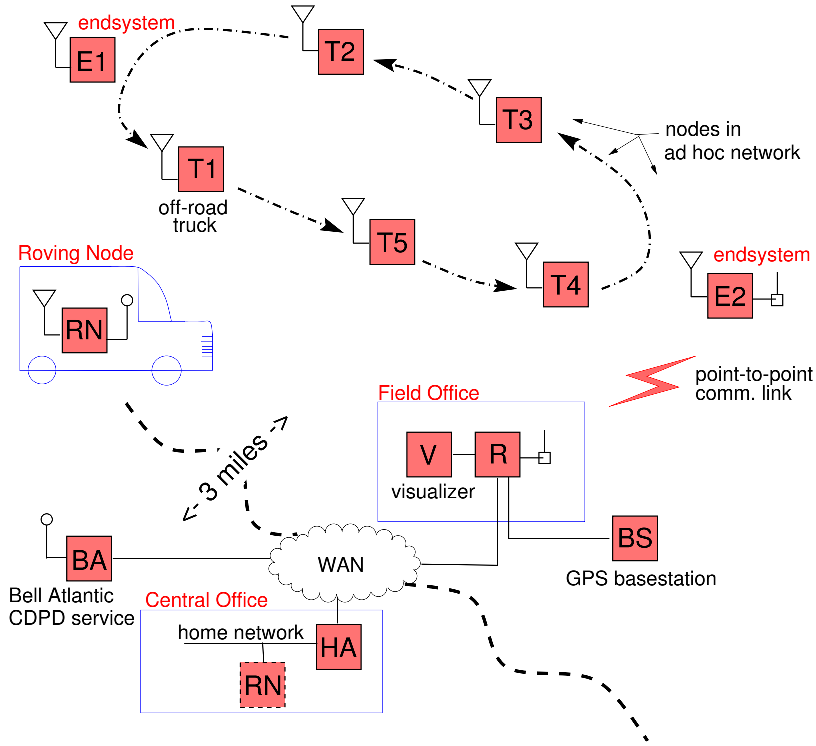

Figure 3.1.: Overview of a real sytem testbed network (courtesy by David A. Maltz)

these mobile nodes, there are base stations and a "roving node" which moves between different networks. The basic layout is depicted in Figure 3.1.

The nodes move on predefined paths so that they are in transmission range of each other and thereby are capable of communicating. During the drive, they exchange data. By varying the number of nodes and the paths, nearly arbitrary scenarios can be examined.

This approach probably is the most realistic way of testing protocols for mobile ad hoc networks. Many characteristics of it are real. For example, properties of the physical layer and the MAC layer are authentic, since the real medium and real network interfaces are used.

This sounds tempting but this approach bears some problems which are discussed in the following.

First, the hardware and the cars needed have to be available. However, the acquisition of new hardware can be expensive. Additionally, an appropriate proving ground has to be found that fits the requirements of the scenario that is going to be examined.

Secondly, management of the personnel is not easy to handle. Enough drivers have to be recruited to carry out the test. Since most of the recruited drivers are not familiar with that topic, they have to be informed in advance ("mission briefing").

Thirdly, reproducibility of the results is not guaranteed. It depends on many factors. The most important factor for reproducibility is the environment during a run, which has to be exactly the same at later runs if one wants to reproduce the results of a previous run. That means, the weather is supposed to be the same, as well as the speed and movement of cars. This is very hard to reproduce, if at all.

Fourthly, wireless communication is unpredictable [MaBJ99]. In the approach developed by the authors of [MaBJ99] communication between two nodes which were in *line-of-sight*, was the poorest. But in theory, this kind of communication is supposed to be the best. Additional influences, that are often omitted by theory or cannot be modeled, have to be taken into account, like reflection and dispersion of signals when encountering buildings, hills or the atmosphere.

But nevertheless, despite all the problems mentioned, this approach can be used as a reference for simulation and emulation results.

3.1.2. Emulation

Emulation is a relatively new research topic. It has a higher level of abstraction compared to real system testbeds. Basically, in emulation is the mimicry of a network configuration other than it really is, i.e. the network pretends to be of a different configuration. The interconnection of network nodes is called a *testbed*.

One advantage of emulation is its ability to run in realtime. As we will see in Section 3.1.3 that does not hold for simulators. Emulation can be divided into three categories, *centralized*, *distributed* and *hybrid* emulation.

Centralized Emulation

Emulators based on the centralized principle use a central server to control the emulation. The server controls all connection parameters and the synchronization during the emulation process. But since all traffic has to flow through this central server, it forms a performance bottleneck.

Seawind [KGMS+01] is an example for this centralized approach. The emulator intercepts all packets sent from a client node to the server. It emulates the desired properties by dropping packets, delaying or by modifying them.

Distributed Emulation

Distributed emulators do not use any central control unit like a server. The emulation takes place on the network nodes themselves. Therefore, appropriate emulation tools have to be installed on the nodes.

Since no central server is used, distributed emulators can handle much larger scenarios with higher traffic rates. Unfortunately, the absence of a central control unit complicates synchronization and control of the emulation. So-called *control packets* have to be used to maintain the emulation. Control packets are used for synchronization purposes in the network. These packets influence the measurements and the network behavior. Therefore, distributed emulators do support dynamic scenarios where the parameters change during runtime.

Examples of distributed emulators are *Dummynet* [Rizz97] and *MobiEmu* developed in [ZhLi02].

Hybrid Emulation

To overcome the deficiencies of both, centralized and distributed emulators, hybrid emulators have been introduced. Here, the configuration of the nodes is done via a central server. The *Network Emulation Testbed* [HeRo02] is an example for a hybrid approach.

3.1.3. Simulation

This subsection deals with simulation in general. First, we discuss the basic characteristics of simulation before we analyze the different types of simulation. Thereafter, Section 3.2 provides a detailed view on simulation of computer networks.

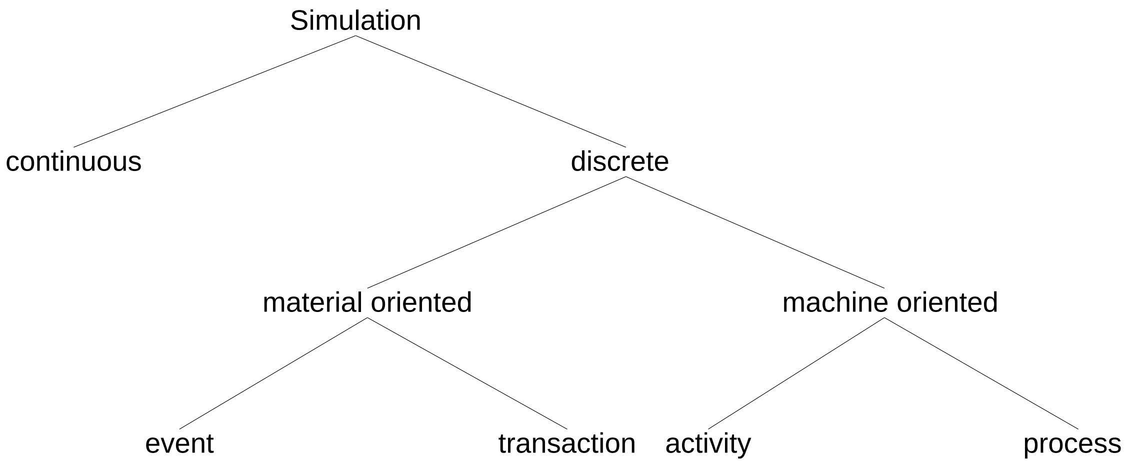

Figure 3.2.: Classification of simulations

According to [Page91] and [Fish01], in a simulation, a *model* replaces the original system during the process of a system analysis and experiments are carried out on this model. A model describes a part of the real world that is of interest. Simulation abstracts by the means of modeling from a real system. Thus, the advantages of simulation become obvious. They advantages are as follows.

Complex systems can be analyzed using simulation, since we concentrate on the relevant parts of the system. The simulator does not need to understand every part of the system in great detail. Additionally, modeling for a simulation increases the modeler's comprehension of the system. We have to think about the system in order to model it. Simulation is an alternative to real-world experiments. As we have seen in Section 3.1.1, the experiments in real system testbeds can be complex and cause some problems, like difficulties in reproducing results. Furthermore, simulation allows to examine the effects that are caused by changes to a system in a simple and inexpensive way. In contrast, real systems often require the acquisition of new hard- or software.

However, simulations also have some disadvantages. The first disadvantage has to do with abstraction. Abstraction typically leads to simplifications of the real system which is reflected in inaccurate simulation results. Therefore, a high degree of realism is desirable. But that may increase the model's complexity tremendously and slow down the simulation. Because of this, a tradeoff between an accurate model and simulation time and complexity has to be found. Secondly, it is costly to develop new models. Thirdly, one must be aware that simulation results may be wrong. Therefore, they require validation, e.g. by using real-world experiments.

As depicted in Figure 3.2, the field of simulation breaks down into two main parts, *continuous* and *discrete* simulation. We discuss the main differences of the various types of simulation with regard to our needs in the following paragraphs.

Continuous Simulation

Continuous simulations rely heavily on mathematics [Crai96]. The simulation components' behavior is typically described by differential equations.

Using these equations, a component's behavior at an arbitrary point of time in simulation can be determined. A simulator uses these equations in compliance to the simulation conditions to create a graph. This graph is used to get information about the state of a component at any point of time. An example of continuous simulation is the simulation of an electric circuit.

However, continuous simulators have some major drawbacks. First, differential equations make this approach computationally expensive. Secondly, due to a lack of knowledge of a component's nature, not all possible components can be modeled using mathematic equations. Due to these drawbacks, continuous simulation cannot be used for our simulator.

Discrete Simulation

Discrete simulation is usually employed for components that have a higher level of abstraction as the ones used in continuous simulation. Discrete simulation basically consists of four *world views* [Page91]: *event-based*, *activity oriented*, *process oriented* and *transaction oriented* simulation. These are classified into two groups, *machine* and *material oriented*.

The material oriented approach is primarily concerned with the flow of materials, e.g. the packets in a network simulation. The machine oriented approach focuses on the processing, e.g. processing of packets a network nodes.

In the following, we discuss the four world views in order to determine which approach fits our needs best.

The first view is *discrete activity oriented simulation*. Here, declarative conditions define the beginning and ending of activities. These activities have to be checked for every activity at each point in simulation time. This is computationally intensive and therefore inefficient. Due to this, activity oriented simulation does not fit our needs.

The second approach, *discrete transaction oriented simulation*, is based on blocks, which are used to model static system components. Transactions are used for dynamic system components respectively. Transactions are characterized by the number of parameters which change during their way through the blocks of the modeled system. Changes in states by transactions are triggered by blocks. This type of simulation is too static to fit our needs.

One of the two major world views is *discrete process oriented simulation*. In this view simulation objects with their associated activities are combined into a *process*. There is a distinction between active phases and inactive ones. Events are shown in the active phases, i.e. state transitions take place in these phases. During an active phase, the simulation time is stopped, i.e. from an outsider's point of view these phases happen instantaneously. The inactive phases are the place and time of calculations. All calculations are made in these phases. As soon as a processing (calculation) is completed the simulator switches from an inactive state to an active, where the results of the processing is presented.

A wide-spread and commonly used technique is *discrete event simulation*. It is based on a detailed description of events by the means of event handlers. Only state transitions that occur during a specific event time are reproduced. Other activities between two events are ignored. This means that processes that need a longer period of time are reduced to a sequence of events. The processing of activities that are associated with an event do not consume any simulation time. The procedure of event processing is as follows. As soon as an event gets active, i.e. its event handler is invoked, the simulation time is set to the time associated with that event. Then the event is processed. After the processing of the event, the simulation time jumps to the next event's time. One advantage of this approach is that longer periods of inactivity, e.g. five minutes in simulation time, are processed instantaneously, i.e. these five minutes are skipped.

Since most simulators rely on discrete event simulation and because it allows very fine-grained simulation, our design uses discrete event simulation. Additionally, several other resources for discrete event simulation exits, like *SIM* [BoEl95], a C++ library for discrete event simulation, which can be used as a reference.

Now that we have described the basic principles of simulation, we focus on specific network simulators in the following section.

3.2. Simulators

Network simulators are a widespread means for testing and evaluating protocols and algorithms. Some of the first simulators go back to the 1980s, e.g. the REAL simulator [Kesh88] and NS-2 [FaVVp04] that was derived from REAL in 1989.

This first generation of network simulators focused on wired networks. With the advent of mobile and wireless networks in the late 1990s, a new generation of simulators was required, because this kind of networks impose more complexity, since network nodes may move and communication is typically more unreliable than in wired networks. Therefore, wireless and mobile scenarios are more difficult to simulate [John99].

During the last few years, simulators were developed that address these problems. On the one hand, there are simulators for wired networks enhanced with wireless features, e.g. NS-2 extended by the Monarch Project of the Carnegie Mellon University [CMU 98] to support wireless and mobile networks. On the other hand, completely new simulators have been designed, which are tailored to mobile and wireless networks. Some of these developments are the "Global Mobile Information System Simulator" (GloMoSim) [BTAT$^+$99, ZeBG98] and its commercial successor QualNet [Scal04], as well as the "Georgia Tech Network Simulator" (GTNetS) [Rile03a, Rile03b].

We discuss some of these simulators in the following and examine which concepts can be used for our simulator.

3.2.1. Network Simulator 2

The *Network Simulator 2*, or *NS-2*, is one of the oldest representatives of network simulators. It is based on the REAL simulator, from which a project was split in 1989. The result of this project was NS-2.

Due to its long existence and a great multitude of protocols and models available for it, it has become the quasi-standard in the field of network simulation. Measurements and evaluations are mostly done with it or at least compared with its results.

NS-2 is focused on fixed, wired networks. It has been extended with properties and models of mobile and wireless networks over the years by the CMU Monarch Project [CMU 98].

Architecture

NS-2 is an object-oriented simulator that uses a split programming model [BEFF+00], i.e. two different programming languages for different purposes. The developers of NS-2 [FaVVp04] identified two primary tasks for their simulator.

On the one hand, it has to support efficient implementation and simulation of protocols. This requires an appropriate programming language that is fast and efficient. The authors of NS-2 have decided to use C++. On the other hand, it has to provide the possibility of changing parameters in a scenario quickly, since parameters often vary between two different simulation runs. Usually, the main part of the simulation, e.g. the protocols used, stays the same. For the latter purpose, NS-2 provides an OTcl interpreter as front-end. OTcl is a scripting language. In NS-2 it is employed to describe and build scenarios.

NS-2's foundation is discrete event simulation, as pointed out in Section 3.1.3. Events are managed in an event queue. An event in NS-2 consists of an unique packet identifier, a time-stamp in simulation time and an associated event handler. As soon as an event is taken from the queue for processing, the simulation time is advanced to the point of time defined by the event's time-stamp and thereafter, the desired action is triggered by invoking the event handler.

This simulator uses *nodes* as basic computing units, e.g. mobile devices or workstation PCs. Nodes are connected with each other by using *links*. Communication between nodes takes only place via the exchange of packets. Various types of links exist, e.g. duplex and simplex links.

There are two possible destinations for a packet on a link. The first one is the actual target node, the second destination is the so-called "drop target", used when the packet is dropped, e.g. due to a transmission error.

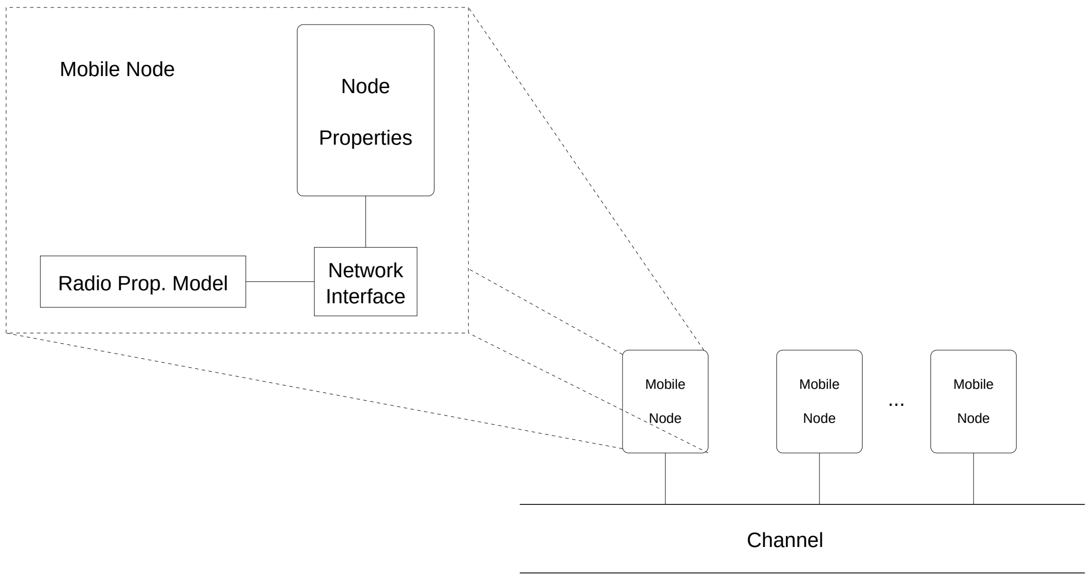

Figure 3.3.: NS-2 – Monarch channel model

The wireless case [CMU 98] is different. Nodes still are independent units. Each node owns one or more *network interfaces*. Each network interface is connected to a *channel*. The channels are used for communication. A channel typically denotes a particular radio frequency with a particular modulation and coding scheme. Packets that are sent on one channel do not affect packets on other channels.

If a packet is sent from one node via a specific network interface, a copy of this packet is given to all interfaces and thereby to the nodes that are linked to that specific channel. Each interface checks the packet by using an associated *radio propagation model* if the packet can be received correctly. This situation is depicted in Figure 3.3.

Enhanced Wireless Support

There are several additional optimizations for NS-2. They improve the performance of NS-2 when simulating mobile and wireless networks. NS-2 with the Monarch extension has severe performance problems when simulating large scenarios with many nodes and lots of traffic [NaGr03].

Naoumov and Gross [NaGr03] demonstrate these problems by simulating a simple scenario with the following configuration: they use 300 nodes, spread over an area of $3 \times 3km^2$. The simulation takes ten seconds of simulation time. The total runtime of simulator is measured at 1000 seconds. This increases dramatically if the duration in simulation time is extended.

Therefore, [NaGr03] developed some improvements, *grid-based* and *list-based node organization*, which result in claimed performance improvements of 5 – 60 %. They show that their variant of NS-2 has linear growth in runtime, whereas the original simulator shows exponential growth.

Another optimization is introduced in [BaSB03]. It enhances NS-2 by analytical models, so-called *analytic network clouds*. The authors of [BaSB03] show that analytic models are faster than pure discrete event-based simulation models. They claim that the performance increase is significant.

Adding Emulation Concepts

As we have seen in Section 3.1.3, simulation bears some disadvantages. Among them are inaccuracies due to abstraction and that no real network traffic can be evaluated by the test implementations of algorithms. This is where emulation comes into account. Various simulators have adopted emulation concepts to overcome the weak points of simulations.

NS-2 is also capable of using emulation concepts [Fall99]. We discuss the basic ideas in the following. There are two basic problems when enhancing simulators with emulation concepts.

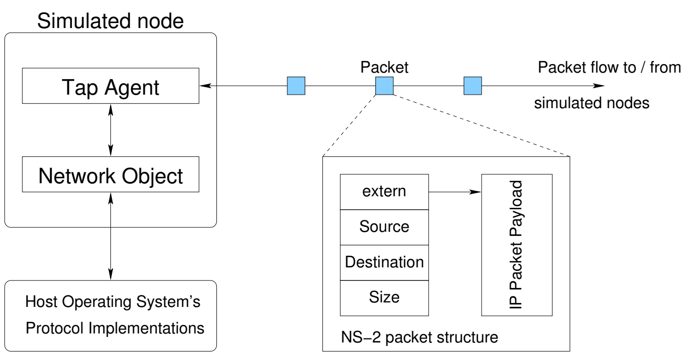

Figure 3.4.: Conversion from network-native packets to simulator-compatible packets

First, the handling of simulation time has to be modified. The real time has to be approximated, since packets from real network traffic enter and leave the simulator in real time. Therefore, real time and simulation time have to be mapped. This is done in the following way. Consider an event that is going to be dispatched at t_1 in simulation time. Then this event has to wait until t_1 is reached in real time. As soon as this happens, the event is dispatched.

Secondly, one has to overcome addressing incompatibilities between the real-world and the simulator. [Fall99] introduces *tap agents* that do the conversion. The basic functionality is depicted in Figure 3.4, which is based on a figure from [Fall99], p. 3. Tap agents are connected to network objects which translate incoming and outgoing packets from the external structure to an internal representation and vice versa.

Evaluation of NS-2

In the previous sections we have described the basic principles of NS-2. In the following, we evaluate them with respect to our needs.

The use of scripts to create and modify scenarios is convenient. But the continuous split programming model bears disadvantages. OTcl slows down the simulator and the insertion of new algorithms into the simulator is made difficult, since both OTcl and C++ code have to be provided. In addition, loading of algorithms during runtime is not possible with NS-2. To achieve that, a recompilation of the whole simulator is required. But this violates one of our major requirements.

Nevertheless, the approach of the Monarch project using multiple network interfaces, radio propagation models and channels seems promising. It allows to model real network nodes in a more realistic way, since they may use different ways of communication, i.e. real-world network nodes typically are not restricted to communicate via one channel and network card.

3.2.2. Georgia Tech Network Simulator

The *Georgia Tech Network Simulator* (GTNetS) [Rile03a, Rile03b] was developed due to deficiencies of NS-2. The authors of GTNetS decided that the removal of these deficiencies in NS-2 is more difficult than the development of a new simulation tool. Therefore, a new simulator was created.

GTNetS is a relatively young simulator, its development started in 2002. It is designed to support very large scenarios. Therefore, the authors state that its primary design foci are *scalability* and *distribution*.

Distributed and parallel computation leads to new challenges, especially concerning network partitions. Computation of routes is a major problem in a distributed environment. Traditional computations of routes utilize global knowledge about the network topology. This is simple to achieve in the non-distributed case, since the simulator's core is aware of the total topology at each point of time. But this does not work in the distributed case, each simulation node or process only has local awareness of the part of the network it manages.

To solve this and other problems, GTNetS introduces *ghost nodes* [RJFA04]. The user who wants to simulate large scenarios typically does not want to log every detail of all components. Therefore, logging in GTNetS can be customized in the simulation script. However, this custom control on logging influences scalability: the more detailed logging is, the more information has to be processed in the simulator and the slower the simulator becomes.

Architecture

GTNetS is fully implemented in C++. It uses an object-oriented design. To create a new scenario, a C++ `main` method has to be implemented, which instantiates all necessary components. GTNetS does not use any scripts to build scenarios, like NS-2 does with its OTcl scripts.

This simulator based on discrete event simulation distinguishes *nodes*, *Interfaces*, *links* and *protocols*.

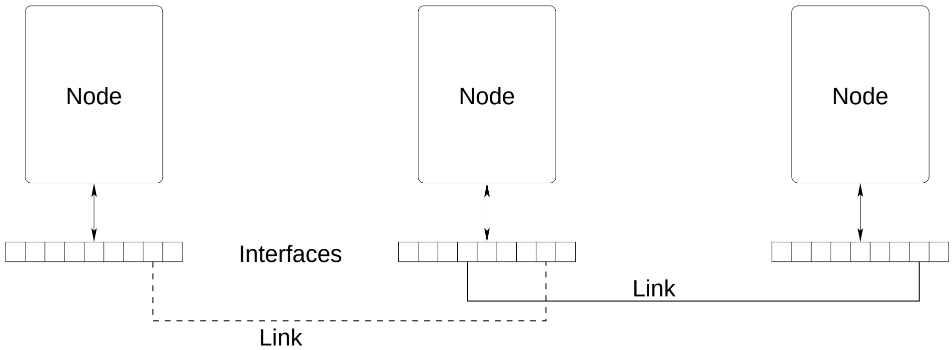

Figure 3.5.: GTNetS general architecture

The basic architecture is outlined in Figure 3.5. Nodes represent actual computing units, like PCs, routers or mobile devices. Communication media in GTNetS are called *links*. A link describes the transmission medium's basic behavior. Interfaces are used to connect a node with its associated protocol stack to link.

A protocol stack is modeled by means of a protocol graph. This graph allows the identification of appropriate protocols when traversing the stack.

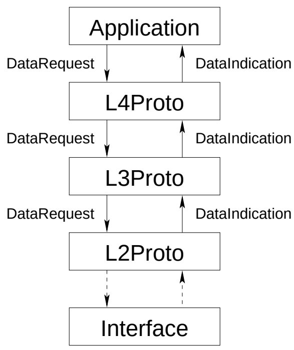

Figure 3.6.: GTNetS protocol stack layout

GTNetS already provides a variety of protocols, but is basically restricted to the ISO/OSI reference model [Tane03]. Protocols in GTNetS are associated with a specific protocol stack layer, i.e. each protocol needs to inherit from one of the protocol stack layer classes, e.g. l4proto. If introducing a new base class, one is able to extend the protocol stack by new layers. The stack layout is depicted in Figure 3.6.

Evaluation of GTNetS

Accounting for the young age of GTNetS, it is a very good and promising simulation tool, mainly due to its design and its approaches for parallel and distributed simulation.

The basic idea of a protocol stack may be a good basis for our design. Unfortunately, GT-NetS neither allows the creation of scenarios via script files which diminishes the flexibility nor changes to the protocol stack and other components at runtime. Thereby, two of our requirements are violated.

3.2.3. Global Mobile Information System Simulator

The *Global Mobile Information System Simulator* or *GloMoSim* [ZeBG98, BTAT+99] is a fast and scalable network simulator that allows simulation of scenarios with several 100000 nodes. It was developed at the University of California Los Angeles. It mainly focusses is on the simulation of mobile and wireless networks. GloMoSim, like the other examined simulators in this thesis, is based on discrete event simulation.

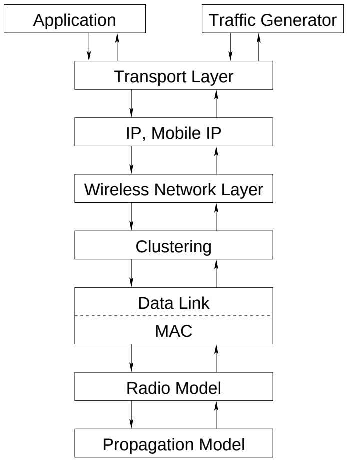

Figure 3.7.: GloMoSim architecture (adapted from [ZeBG98])

GloMoSim provides a set of libraries that can be used to model a simulation scenario. It can be used as a sequential simulator, but it focusses on parallel simulation. Therefore, it is implemented in a special programming language, PARSEC (Parallel Simulation Environment for Complex Systems).

In 2001 it was enhanced and introduced as a commercial product, QualNet [Scal04]. In addition to GloMoSim, among other things, QualNet supports the use of complex environmental models. As stated by the developing company, Scalable Network Technologies, the support of mobile ad hoc networks is significantly enhanced in QualNet compared to GloMoSim.

In the following subsection, we discuss GloMoSim's architecture in greater detail, and focus on the design of the protocol stack.

Architecture

The architecture of the protocol stack is of particular importance in GloMoSim [ZeBG98]. It is composed of a layered structure. Figure 3.7, based on a figure from [ZeBG98], depicts this layered structure.

A great variety of models have already been implemented for each layer. If we take a glimpse at the propagation models for example, we can identify three different models: free space propagation, a fading channel model and an analytical model.

Two models are interconnected using APIs. The APIs resemble those for building a protocol stack [Tane03]. Consider Figure 3.8, where the *network layer* is connected with the *data link layer*. This interconnection is realized by message exchange via two particular methods.

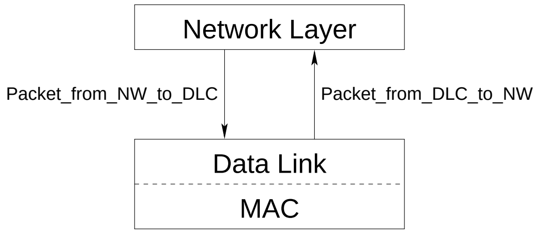

Figure 3.8.: Connection of network and data link layer

Each protocol model has to comply to the API of the layer it is implemented for. But this restricts the development of protocols to the layers GloMoSim provides. Nevertheless, by strict compliance to the APIs, an exchange of protocol models is possible.

Communication between simulation entities takes place by transmitting time-stamped messages, in line with the discrete event simulation principle.

Evaluation of GloMoSim

As shown in many experiments and papers, GloMoSim is a fast and flexible and fully matured network simulator. But it has some drawbacks. First, the use of a special programming language. Although PARSEC is similar to C, one needs to learn this new language and one has to re-implement the software under test in PARSEC. The second and most important drawback is that the development of GloMoSim is no longer continued, since the focus is on its commercial offspring.

3.3. Summary

We discussed related work in the field of network simulation in this chapter. First, we presented the three basic types of performance analysis: real system testbeds, emulation and simulation. Thereby, we discussed the problems that arise with real system testbeds, e.g. reproducibility. We decided that discrete event simulation is the type of simulation that fits our needs best, since continuous simulations require complex computation, e.g. solving differential equations.

After this, we examined and evaluated three representatives of network simulators: NS-2, GTNetS and GloMoSim. We introduced the concepts that may be adapted for our simulator. NS-2 uses configuration files written in OTcl to describe scenarios which results in greater flexibility than the implementation of the `main` method, as in GTNetS.

However, GTNetS is more flexible regarding network interfaces. It allows the user to add more than one network interface to a node which increases the fields of application for nodes.

GloMoSim provides an efficient way of combining algorithms in a protocol stack by using APIs. However, these APIs are hard coded and do not allow arbitrary combinations of algorithms, i.e. there is no way to change the structure of the protocol stack.

4. Architecture

In this chapter, we describe the general requirements and goals of this diploma thesis and discuss the requirements for our simulator. Based on this, we develop an overall architecture for our simulator. We also identify the components that are subject of this thesis.

4.1. Requirements

As we have seen in Chapter 3, numerous networks simulators exist already. Most of them are specially designed and optimized for wired networks, e.g. NS-2 [FaVVp04, McFl97]. Additionally, simulators for wireless networks do exist.

The increasing dissemination of mobile devices enables the gathering of huge amounts of data. This requires efficient distributed algorithms for processing this data. However, before being deployed in real systems, these algorithms have to be tested and evaluated. As we have seen in Chapter 3, network simulators can be employed for this evaluation and testing process.

These simulators support the development process of such algorithms. Unfortunately, the state-of-the-art simulators have some deficiencies that negatively affect the development process. We outline major deficiencies and derive the requirements for our simulator from them.

The first deficiency is the lack of interactivity in current simulators, i.e. changes to the simulation scenario during runtime are hardly possible, if at all, in these simulators. Secondly, simulators like NS-2 do not scale properly, i.e. the simulation of large scenarios with many nodes and lots of network traffic is difficult and slow [NaGr03].

Thirdly, most simulators are not modular. For example, it is not possible to load algorithms into the simulator during runtime. In fact, simulators typically require algorithms, that are supposed to be tested, to be included at compile time. However, this lack of modularity bears a problem. Each user has to compile the complete code of the simulator in order to commit changes or new algorithms to the simulator. This increases the risk of errors in the simulator.

Fourthly, simulation experiments have a high resource consumption regarding CPU time and memory. One reason for this is that some simulators like NS-2 always do some calculations, e.g. the calculation of paths between nodes, although the calculated information is not relevant to the current simulation.

We are able to identify our requirements based on the deficiencies of other simulators as follows. Our simulator has to provide interactivity in order to enhance flexibility and adaptability of a simulation tool, i.e. the user is able to modify simulation relevant parameters and influence the simulation during its run.

Another important requirement is a modular overall design of our simulator. It shall be possible to load components during runtime and adding new components to our simulator shall not impose the need of compiling the whole simulator again.

Additionally, our simulator is supposed to have small resource needs, e.g. reduced memory consumption. Whether our simulator has high resource consumption or not, relates to its capability of handling large simulation scenarios with many nodes and lots of network traffic.

4.2. Cubus Architecture

We present the basic architecture of our Cubus simulator in this section, briefly pointing out the single components. The components of the overall architecture, as depicted in Figure 4.1, are subject of two diploma theses.

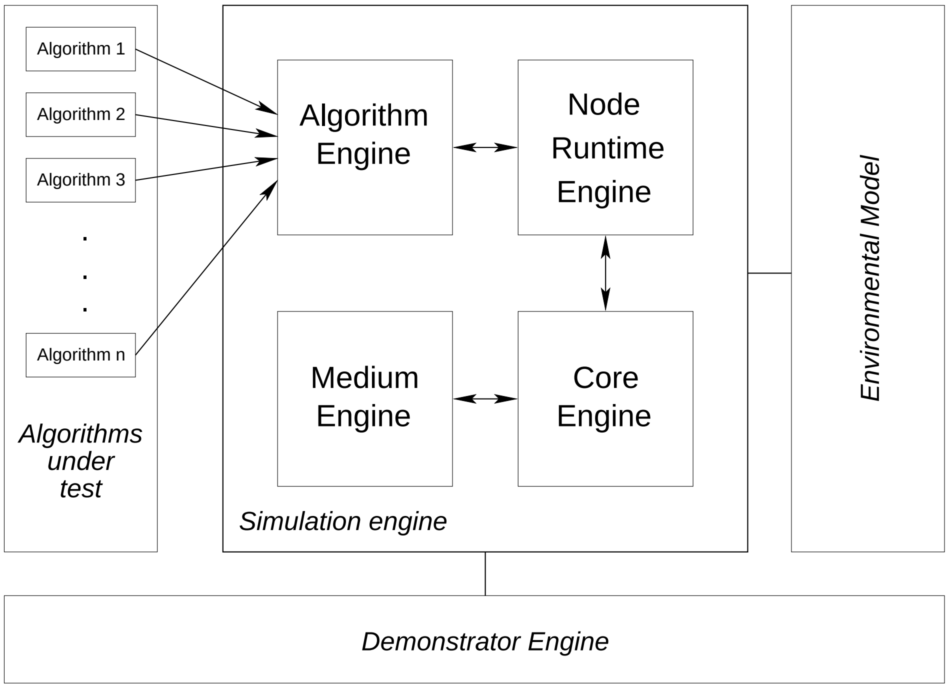

Figure 4.1.: Cubus Architecture

While focusing on the components relevant to this diploma thesis, the remaining components are mentioned briefly because they directly affect the design decisions.

The *algorithm engine*, the *node runtime engine*, the *medium engine* and the *core engine* form the central components of our simulator. They are combined in the *simulation engine* which is responsible for carrying out all basic tasks necessary for running a simulation. It manages all objects in a simulation and their interaction.

Both the *environmental model* and the *demonstrator engine* extend our simulator by adding new features like mobility of network nodes and interactivity.

The simulation engine and its components are subject of this thesis. However, the *environmental model* and the *demonstrator engine* are part of a second thesis that currently is in preparation. Therefore, we only briefly discuss the environmental model and the demonstrator engine. The design of the core engine is described in Chapter 5 and the design of the remaining parts of the simulation engine is discussed in Chapter 6.

4.2.1. Simulation Engine

As we have seen, the simulation engine comprises the central components of our simulator. Its main parts are the *Algorithm engine*, the *Node runtime engine*, the *Core engine* and the *Medium engine*. The engines provide a conceptual view on their underlying parts. They can be considered as a common interface of the underlying simulation components like nodes and algorithms.

Algorithm Engine

Algorithms are entities that describe specific functionalities, e.g. they provide routing mechanisms. The algorithm engine defines the basic capabilities and properties of algorithms by providing a common interface for them. It is possible to exchange algorithms and change their behavior during runtime. Due to this, algorithms are dynamically loadable during runtime.

Algorithms form a basic unit of our simulator. They can be connected with each other to build a protocol stack of a node. By this means the ISO/OSI reference model can be constructed. But our simulator is not restricted to the seven layers of that reference model. In fact, an arbitrary number of layers is supported.

Node Runtime Engine

Algorithms are connected with each other and thereby form a protocol stack. Each node is associated with its own, unique protocol stack. Nodes are important simulation components. Nodes represent basic computing units, like workstations or mobile devices. Conceptually speaking, the node runtime engine provides a common interface for all nodes and thereby defines their basic layout.

Nodes are flexible and adaptable simulation components that can be customized by attaching other components, e.g. positioning devices.

Core Engine

The core engine forms the simulator's core component. It carries out all central tasks of our simulator. It is responsible for coping simulation events, controlling the simulation time and keep the simulation running. In other words, it forms the heart and brain of every simulation. Most components of our simulator are connected via the core engine. New components can be plugged to this engine in order to extend the simulator by providing new capabilities.

Medium Engine

Media are the components via whom communication between nodes is carried out. The real-world counterpart of media are the actual physical media that are used for data exchange, e.g. wires in case of wired networks and air in the case of wireless networks.

Compared to the algorithm engine and the node runtime engine, the medium engine provides a common interface for all media. Our simulation media models the physical characteristics (error rates, . . .) of their counterparts.

Media are attached to protocol stacks. Using this connection, nodes can transmit packets to the medium and thereby communicate with other nodes. We refer to point-to-point links, if only two protocol stacks of different nodes are interconnected with each other. otherwise we speak of shared medium.

4.2.2. Environmental Model

Real-world wireless networks typically use some kind of mobile devices. These devices move in specific environments. These environments can be very complex. The environmental model extends the functionality of our simulator by employing real-world characteristics like mobility of nodes and obstacles in simulation areas. Therefore, without the environmental model, simulation of mobile networks is rarely possible, since the simulation engine lacks the possibility to use positions and mobility. The environmental model comprises the *location service*, the *spatial model* and the *mobility engine*.

Location Service

The location service manages the exact positions of each simulated node. These positions reflect the notion of real-world physical coordinates of nodes. These exact positions may be falsified in special positioning devices that are attached to a node. This falsification can be used to model the inaccuracies of available positioning devices like GPS receivers. Additionally, in wireless networks the location service is used to determine whether two nodes are in transmission range of each other and therefore can communicate.

Spatial Model

The spatial model adds obstacles like buildings to the simulation area, i.e. the area where all nodes are placed and communicate. Without the spatial model, the simulation area consists of a plain and empty field which only contains the simulated nodes. But this is far from being a realistic model. The introduction of spatial models overcomes this problem.

The spatial model is responsible for the determination of the reachability of nodes and effects on signal propagation. For example, if a building is between two nodes, it is rarely possible for them to communicate with each other. Thereby, the spatial model confines the communication possibilities of nodes by adding obstacles.

Mobility Engine

Up to now, only fixed positions of nodes are possible. To simulate movements of nodes, mobility models are necessary. The interface of mobility models is defined by the mobility engine. Mobility models are associated with each mobile node and can be exchanged or modified during runtime. This increases the flexibility of our simulator.

4.2.3. Demonstrator Engine

The demonstrator engine adds interactivity and adaptability to our simulator by providing new components. The demonstrator engine mainly consists of the *analysis engine* and the *interactive engine*.

Analysis Engine

The analysis engine is used to examine specific parameters and events of the simulation in detail. Since simple text output of logging information is not sufficient in many cases, this engine supports various output formats. This can be a text file but formats like relational databases are possible.

Interactive Engine

The interactive engine acts as a bridge between its use as a simulator and as demonstrator. It is attached to the simulation engine and is used to visualize the simulation. Thereby, it provides the possibility to interactively change parameters of the simulation during runtime. Therefore, the interactive engine is a powerful and flexible tool.

5. Design of the Core Engine

This chapter describes the core engine of the Cubus simulator. It comprises the simulator's basic functionalities, ranging from registration processes to event handling.

First, we will briefly show that our simulator is based on discrete event simulation and why this approach was chosen. Secondly, we will introduce events which are used in Cubus as the fundamental units of our simulator. Thirdly, a closer look towards event handling is taken, i.e. we will discuss processing of events and the receivers of events. And finally, we will outline the process for setting up a simulation scenario can be set up.

5.1. Some Remarks on Time

Using a simulator requires a clarification of the notion of time [Page91, Fish01]. Several "times" exist in a simulator and are used by it. Basically, we differentiation in *wall clock time* (WCT) and *Simulation Time* (SimTime). This is also the distinction in Cubus.

Wall clock time is the actual physical time elapsed between the start of a process and its termination. It resembles our everyday time. Consider a clock hanging on the wall behind. The time that is shown on it, more precisely, the time between the first look and second look at the clock, recreates the WCT.

SimTime is an artificial construct. It may vary in progress, i.e. non-linear "jumps" in time are possible. SimTime can run faster or slower than wall clock time. It reflects the time the simulation would take, if it was a real-world process.

5.2. Discrete Event Simulation

Basically, there are two choices for designing a simulator. The first possibility is using a continuous model. The second one postulates a discrete model. We have decided to incorporate a discrete model in our simulator, since a discrete simulation has some significant advantages over a continuous simulation (see Section 3.1.3).

Therefore, Cubus is designed as a *discrete event simulator* (DES) [Page91, Fish01, Ball96]. As described in Section 3.1.3, a DES manages events and times. In our case, time is primarily *simulation time* (SimTime). The simulator manages a queue which stores the events according to their simulation time. Events are taken from this queue and thereby other events or actions are triggered.

Cubus uses this property. It stores events in queues (see Section 5.3.3) and processes them. According to [Ball96], there are two methods of handling time in a discrete simulation environment, *time slicing* and *next event*.

The time slicing methods has the advantage that progress is done in constant time intervals, e.g. ten seconds. These time intervals are measured in *Wall Clock Time* (WCT). However, this method bears some disadvantages. The main disadvantage is the same that was mentioned as an advantage: the constant time intervals. These intervals must always elapse before something new can happen. Consider a case where nothing happens in the simulation during such an interval. Anyhow, these interval has to elapse in the same speed, e.g. ten seconds, as if there was much activity. This may slow down the simulation.

A solution to overcome this problem is the next event method, where SimTime is always advanced according to the next event's time-stamp. For example, consider the case where nothing happens in the simulation for five minutes in SimTime. The simulator advances the time to the end of these five minutes in one single step.

This speeds up the simulation, but may cause other problems, e.g. when using *graphical user interfaces*, since jumps between significant points in time may occur. These jumps may be distributed unevenly. Therefore, it can be a problematic, to visualize simulation results. But this goes beyond the scope of thesis.

Taking advantage of experiences with other simulators, like GTNetS [Rile03a] and NS-2 [FaVVp04, BBEF+99], our simulator uses the next event method.

Discrete event simulation is mainly based on events and their processing, as stated previously. Thus, we discuss events in the following section.

5.3. Event Handling

This section describes events and their handling. First, events are defined. We will provide information how they are modeled and what they can do and what they cannot. Secondly, this section covers the process of event registration. Thereafter, since components are registered for specific events, their processing is described.

5.3.1. Events

Events are the basic unit of discrete event simulation. A simulator encodes specific actions into specific types of events. These are scheduled in a queue, e.g. a central queue (see Section 5.3.3) and finally executed. Once executed, events can trigger further events and thereby, keep the simulator running and accomplish all simulation tasks.

To keep our simulator as flexible and adaptable as possible, events need to allow as much freedom as possible to adapt them for specific purposes. Therefore, they may only include basic information, that is common to all possible types of events. A minimum requirement are an unique identifier, a time-stamp in SimTime, a priority and an associated handler. More information is not included in a Cubus basic event.

The identifier is used to identify each event during a simulation unambiguously. The time-stamp tells the simulator when to execute or trigger the event. The handler is responsible for the execution of actions associated with the event. The priority defines which event has to be used if two events have the same time-stamp.

This basic event, with all its minimal requirements, forms the *Event base class*. All other specialized event types inherit from this class. For example, events like *Timer events* or *Log events* may be derived from that base class.

5.3.2. Event Registration

The Cubus simulator employs a *publish / subscribe* system where components can register for specific types of events. They are informed whenever an event for which they have registered occurs.

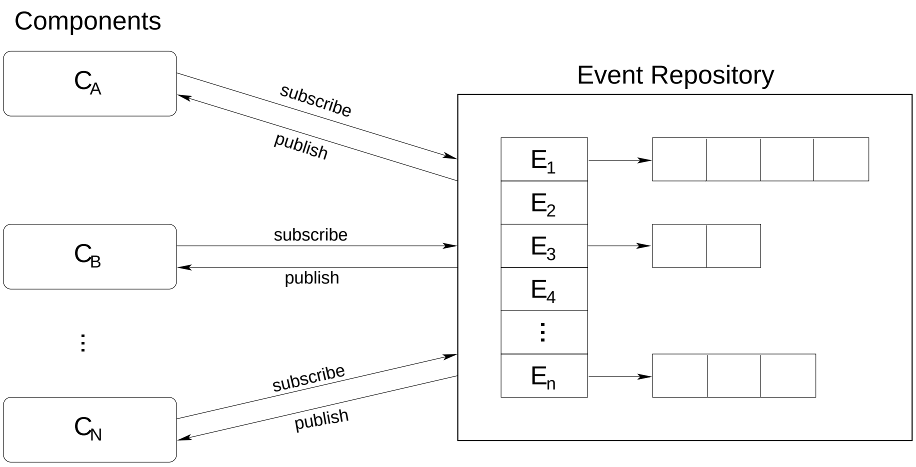

Figure 5.1.: Event repository and publish / subscribe mechanism

The subscriptions are stored in a central *event repository* (ER) which is located in the core engine. It manages a list of all available event types. The user can add new types or remove old ones during startup or runtime.

Components subscribe or unsubscribe for specific event types. In the former case, this subscription is added to the list of subscribers for this event type. This is depicted in Figure 5.1. In the latter case, this entry is removed. Thus, a relation between the subscriber and the event type is created. The simulator knows, that if a specific event type occurs, it has to inform the subscribers for this event. This is called *publishing*. The handling of the event is then realized by the event handler associated with this event. This is illustrated in Section 5.3.3.

Figure 5.1 depicts the basic layout of the event repository. On the left side n arbitrary components (C_A to C_N), are shown. All of them can subscribe for events and get informed, when they occur. Within the event repository there is a list of event types (E_1 to E_n). Each type has an associated list of subscribers. This is shown by the boxes behind them. For clarity reasons only every second row of boxes is visible. Each box represents a subscriber.

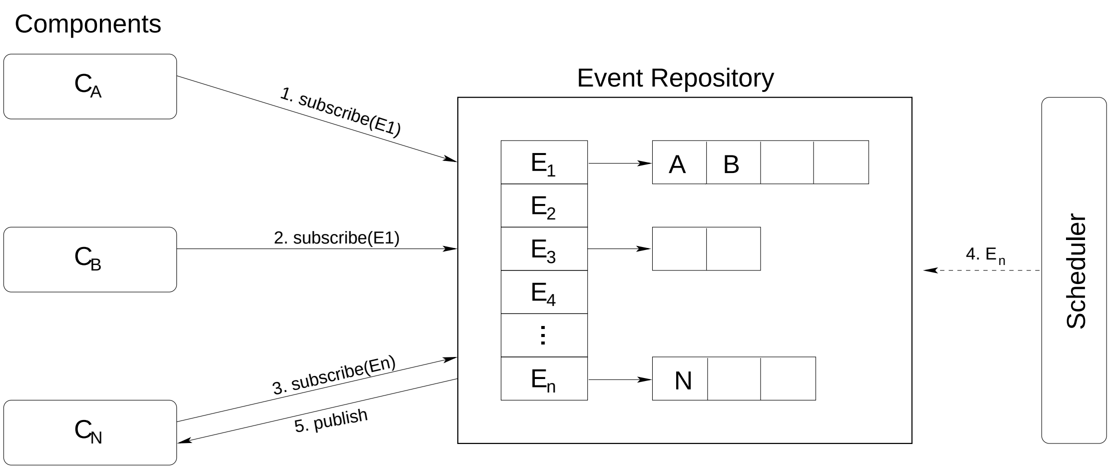

Figure 5.2.: Publish subscribe processing

A simple process sequence is shown in Figure 5.2. At first, the components subscribe for events. Here, C_A subscribes for event type E_1 just as C_B does. Component C_N subscribes for E_n. Then, at step four, an event of type E_n occurs, i.e. it was selected by the scheduler to be the next event to be processed. The event repository is informed. It looks into its list of subscribers whom registered for E_n. Thereby, it comes across C_N. This causes the event to be published to C_N where it is processed. Thereafter, the handler of the event is executed and the action associated with this event carried out.

This publish / subscribe system has a couple of advantages. First, it is a flexible solution. Components can subscribe for events or unsubscribe during runtime. That provides the possibility to change simulation components ad hoc during runtime without the need of restarting the whole simulation.

Secondly, multiple components can subscribe for the same event type and get informed when such an event occurs. This is hard to implement without a central repository for events. It would require multiple synchronized copies of the same event in the event queue. Using a repository only requires one. Assume that the user wants to add multiple *log engines* to the simulator, e.g. one for plain text output and one for database oriented processing. They all have to monitor each log event, but treat them differently. As soon as a log event occurs, copies of that event can be distributed to the log engines.

5.3.3. Event Processing

The actual processing of events is handled by the *scheduler* and *event queues*. First, we discuss the queues used and how events are scheduled. Then, information on execution of events is provided.

Event Queues

GTNetS [Rile03a] and NS-2 [McFl97] basically employ one central event queue for processing events. These queues typically are based on priority queues and in some cases on other sorted data structures.

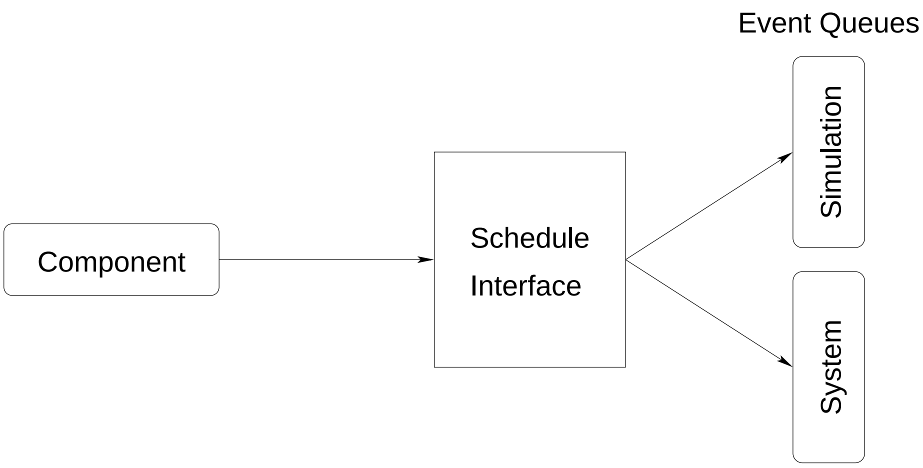

Figure 5.3.: Schedule interface to hide event queues

Our simulator enhances that. As depicted in Figure 5.3, it uses two queues instead of one, the *simulation queue* and *system queue*. We distinguish between *simulation events* and *system events*.

Simulation events are all events that are necessary for proceeding with the simulation, i.e. the events that are conducted for further simulation flow. This includes events for transmitting packets (`PACKET_TX`), for receiving them (`PACKET_RX`) and many more.

System events are those events that are generated during simulation but do not influence the continuation of this simulation. Log events belong to that category. These are required for analysis purposes and for monitoring the simulation, but do not affect the simulation.

The simulation queue stores all simulation events, whereas all system events are processed by the system queue. This unburdens the simulation queue from unnecessary events, which are not required to keep the simulation going.

Both queues work in parallel, which increases the simulator's runtime performance. The system queue may run asynchronously regarding the simulation. Since system events do not influence the simulation, they need not be processed synchronously with the rest of the simulation.

Scheduling of events is done using the *schedule interface*, as depicted in Figure 5.3, which dispatches an event to the appropriate queue.

As mentioned above, the queues are based on priority queues. New events are scheduled according to their time-stamp and their priority.

Event Execution

After events have been scheduled, they have to be retrieved from one of the queues and executed. This is done using *handlers*. Every event has an associated handler.

When an event is taken from one of the queues, the first thing that has to be done, is to update the SimTime. The new SimTime is given by the time-stamp of the event which is currently processed. After updating the SimTime, the event is passed to its handler where it is processed further. After the handler has finished, the next event is taken from the queue and processed in the same way as described previously.

5.4. Management of Custom Parameters

As described in the requirements in Chapter 4, our simulator has to be flexible and adaptable. The adaptability refers to the needs of the user without the need of recompiling the whole system after every change.

Therefore, Cubus supports the possibility to load specific components dynamically during runtime, e.g. algorithms or media (see Chapter 6 for more details). In contrast, almost all of the network simulators examined in this thesis, except OMNeT++ [Varg01], allow only components to be integrated into the system during compile time. Subsequent changes potentially require recompilation the whole system. Depending on the overall size of the system, this may require long periods of time.

Additionally, a simulator which does not support dynamic loading uses more memory at start up time, since a great amount of components has to be loaded into memory at this point in time. In contrast, a system using dynamic loading during runtime allocates memory only for components required at the moment. This keeps it smaller and handy.

But a flexible system results in a different exposure to user-defined, custom parameters. Information about all available parameters of a component is needed. It is necessary that this information can be changed during runtime, e.g. if a new component is loaded or an old one removed.

Therefore, a basic *Container* class exists which is able to store and manage all parameters of an object type. Basically, each object includes two types of parameters. The first one consists of parameters that are common to all objects of this class, so-called *predefined* parameters. The second type consists of custom parameters that can be freely defined by the user. These two parameter types have to be stored in the container for each type.

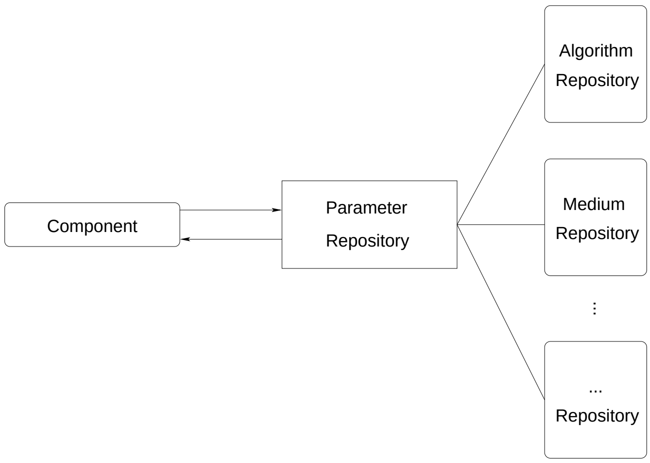

Figure 5.4.: Parameter repository and custom parameters

Class specific containers, called *repositories*, inherit from this container superclass. As depicted in Figure 5.4, there is one repository for each dynamically loadable class, e.g. an *algorithm repository* and a *medium repository*.

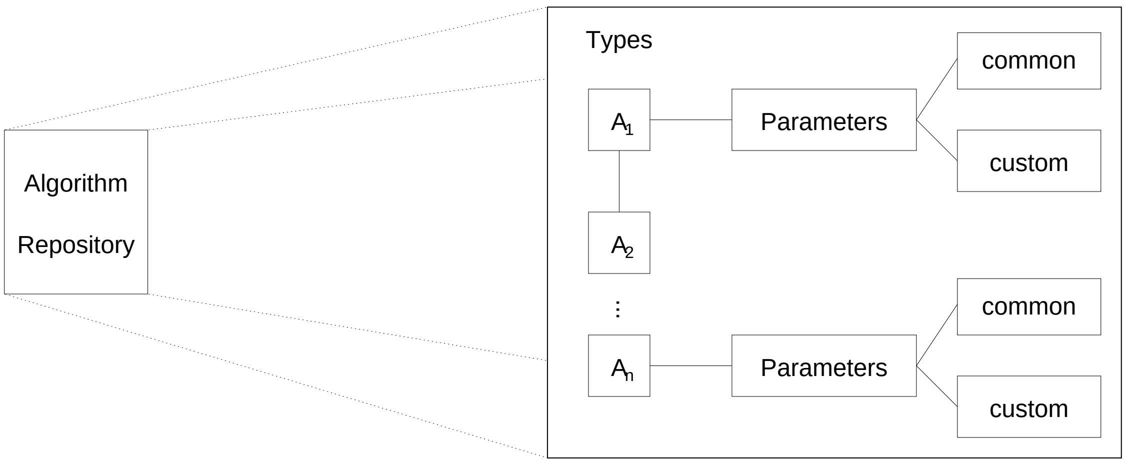

Figure 5.5.: Detailed look into an algorithm repository

Figure 5.5 shows a more detailed look into a special container, namely an algorithm repository. The repository manages a list of all algorithm types ($A1$ to A_n) and relates them to information about their parameters. As mentioned above, the parameters are split into two categories: the common or predefined parameters and the custom ones. This is illustrated by the two different boxes in Figure 5.5. For clarity reasons, the parameters of A_2 are omitted.

The custom parameters include properties which describe their increment, range of values, e.g. the value of a parameter "speed" is restricted to values between one and ten, the speed can be increased by only one in a single step. An arbitrary amount of new properties can be added by the user. Each parameter has its own default value. These default values are used to initialize newly created instances.

If a new class of objects has to be added to the simulator, which is supposed to be dynamically loadable, a new repository is required. The simulator must have access to the parameters available.

To provide a unified interface to other components, a *Parameter Repository* is introduced, as depicted in Figure 5.4. It hides the different repositories from the component. The parameter repository is based on the *facade pattern* [GHJV96].

The unified interface provides access to all available parameters with all their properties of a specific object type. Such information is required if one wants to know which parameters exist and how they can be changed. If a query like "give me all parameters of the simple flooding algorithm" is invoked, the parameter repository analyzes the query and forwards it to the appropriate repository (here: algorithm repository).

5.5. Scenarios

In the previous sections, we have seen the basic functionalities of the core engine. Event handling and the use of user-defined parameters was explained. But the core engine also takes on the assembly of simulation scenarios and the processing of information about dynamically loadable objects, like the custom parameters described in Section 5.4.

This processing is basically carried out by reading *configuration files*. However, the build-up of scenarios can be done in multiple ways (see Section 5.5.2).

In the following, we show the different types of configuration files, describe their functionality and discuss their advantages and disadvantages. After that, the different ways of how to build-up scenarios are pointed out.

5.5.1. Configuration Files

Most of the simulators available basically support one of two ways for building and configuring simulation scenarios. The first way is the use of configuration files, the second one is to implement the scenario in an appropriate programming language directly by invoking methods provided by the simulator.

NS-2 uses the first approach, i.e. it uses configuration files. Scenarios are described and implemented using the OTcl scripting language outlined in the NS-2 manual [FaVVp04].

GTNetS uses the second approach. The user implements the *main* method of the simulator, where he defines and builds the simulation, e.g. creates nodes and links them. After writing this method, it is necessary to compile and link it to the simulator library, the GTNetS core.

The use of implementations like in GTNetS has certain advantages. First, the user does not have to learn a new scripting language which is used by the configuration files, like OTcl in NS-2. Secondly, the interaction of simulation components is understood better than with the use of configuration files.

However, the second approach bears some disadvantages, too. A concrete implementation of the scenario by using the methods provided by the simulator is potentially far more complex to write and understand than using more simple scripts. Furthermore, changes

```
property {
    name = ...;    // the parameters name
    step = ...;     // description of this parameter's properties
    ...
};
```

Listing 5.1: Parameter property construct

in the scenario enforce a recompilation in most cases. This is more time consuming than reloading the simulator with a new script.

On this account, the design of Cubus primarily follows the first approach, the use of configuration files. Nonetheless, other ways exist, as will be pointed out in Section 5.5.2.

There are two types of configuration files for Cubus. The first one defines the parameters of dynamically loadable objects. The second type describes the global scenario file. By using the global scenario file, the scenario which is going to be simulated is built and started.

Both types of configuration files are discussed in the following subsections. The basic rules for writing these files is shown in the Appendix A, where the grammar of the parser is given. For illustration purposes some examples are given there, too.

Configuration Files for Dynamically Loadable Objects

Cubus supports dynamic loading of objects during runtime. These objects can have an arbitrary internal structure, as long as they are conformant to the base object's interface. However, the simulator has to have about the internal structure of these objects and how to access them.

Since C++ does not provide reflection like Java [McCl98], another way to obtain this information is required. This is where the configuration files come into account. They are loaded during startup time to configure dynamically loadable objects. In addition, they are loaded each time a new object is loaded during runtime, which is yet unknown to the simulator. They describe the object's internal structure and provide default values for its parameters.

The description includes the object's *name*, *type* and *location*. The simulator uses the name to identify the object. Its location is mandatory for loading the object. The location tells the simulator where the object can be found, if it needs to load the object. Additionally, an arbitrary number of custom parameters, which are used in the object, can be described. They are provided by a `property`-construct, as depicted in Listing 5.1.

The information gathered while parsing one configuration file are stored in the repository (see Section 5.4) related to this object. Therefore, they are available to and accessible for all other components of the simulator.

Scenario File

The *global scenario file* describes the whole simulation scenario at the beginning of the simulation. Besides general properties like a global log level (see Section 5.6), the simulation components are built and initialized.

Network interfaces, media and much more are defined and declared in the scenario file. Finally, the nodes in the simulation are created, together with their internal structure. A part of the internal structure is the protocol stack, which can be easily connected. After that, the nodes can be interconnected using network interfaces and media. Additionally, events can be defined that start and may influence the simulation. Some examples can be found in Appendix A.3.

5.5.2. Ways of Building Scenarios

The use of scenario files, which is where the primary support of Cubus is focused regarding the building of scenarios, offers advantages but also some disadvantages. One key advantage is the simple creation of scenarios and the possibility of changing them quickly. A disadvantage is the need to learn the syntax for the configuration and scenario files. This situation can be improved through automatic scenario editors.

Therefore, our simulator supports one additional way of creating scenarios. It is the implementation of a main method similar to GTNetS. There is no need for learning a new syntax, however C++ knowledge is required. In this case the correct data structures of Cubus have to be used and supplied with parameters.

5.6. Logging

Logging is one of the key features of every simulator. Log files are necessary to analyze and trace the simulation. They are different for different types of simulation, since each one requires different things to be logged and analyzed. Potentially, every simulation requires different information to be logged. Therefore, logging must be as flexible and adaptable as possible.

Logging in Cubus is determined by *log levels*. Each level supports a different set of data to be logged. Basically, there is one *global log level*, which is predefined by the scenario configuration file. It holds for all components used in the specific simulation. The global log level is regarded as the default log level.

Additionally, there are *local log levels* unique to each component. They are defined by the corresponding component's configuration file, but can be changed later on. If no local log level is specified for a component, the global log level is used.

Special events, so-called *log events*, are used for logging. They are based on the basic event class and are freely configurable. Log events are stored in the System Queue of the simulator (see Section 5.3.3) which runs asynchronously with the simulation. Therefore, the Simulation Queue is freed from unnecessary events, which are not required for going on with the simulation and do not affect the SimTime.

Arbitrary components can subscribe for log events. As soon as a log event arrives, it is published to the subscribers. The advantage is that multiple different "loggers" can be used, e.g. one simply puts the log messages on the standard output while another one stores the information in a database.

Primary subscribers are the *analysis* and *interactive engine*, that need log events for their work. However, two components are beyond the scope of this thesis.

For flexibility reasons, the logging process is committed to the developer of components, e.g. algorithms. The developer is able to define the local log level for the component. This local log level is used to select whether a log event is triggered or not, e.g. a log event is triggered if a log level of three is defined. By using this method, the developer is able to determine the log granularity by himself. This leads to a greater flexibility in use and analysis of simulation data.

5.7. Summary

This chapter described the design of the core engine of our simulator. We have seen that events are the basic units in our simulator. They are used to control the simulation. We introduced a basic event class which contains the minimum number of parameters mandatory, e.g. an identifier and an event handler.

The events need to be processed in some way. First of all, simulation components may be interested in specific events, e.g. a graphical user interface wants to get informed in node movement events. Our simulator provides a publish / subscribe system where components can subscribe for specific event types and get informed whenever an event of this type occurs in the simulation.

The actual processing of events is done using event handlers and queues. Cubus employs two queues: the simulation queue and the system queue. A scheduler stores events in the appropriate queue and decides the next event to be processed. Whenever an event is taken from the queue, the simulation time is updated and the associated event handler is executed. This handler processes the event.

Additionally, we discussed the management of custom parameters for dynamically loadable objects. We introduced repositories for storing information about the object's custom parameters.

Thereafter, we described the use of configuration and scenario files. Cubus requires configuration files for handling dynamically loadable objects. The scenario file is used to build a simulation scenario that is supposed to be simulated.

6. Design of Simulation Components

The previous chapter described the core engine and its components. As we have seen, the core engine forms the central part of our simulator which is responsible for carrying out most management tasks during a simulation, e.g. processing of events.

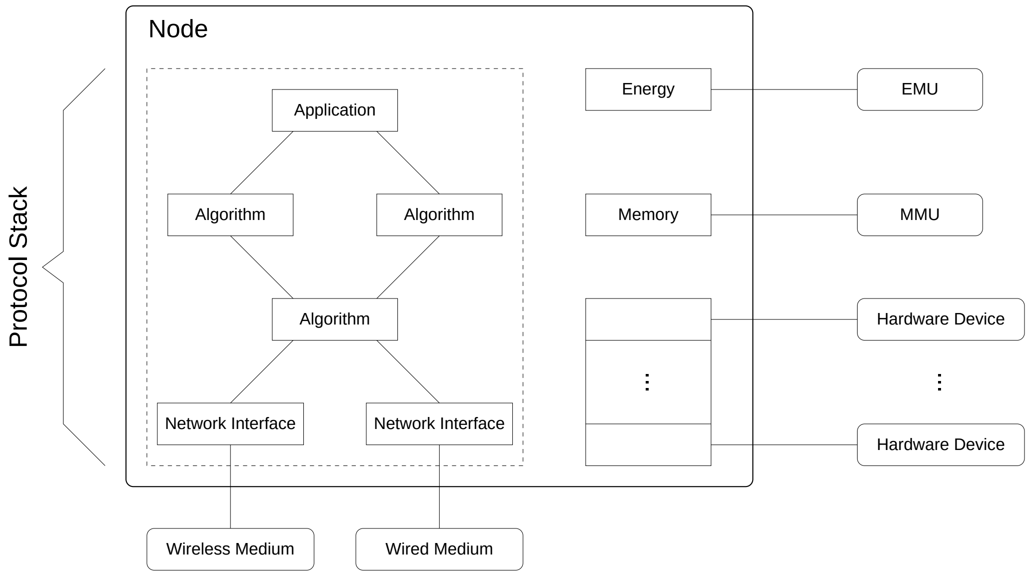

Figure 6.1.: Overview of simulation components

However, this chapter explains the design of the simulation components. These components comprise a simulation scenario and therefore range from algorithms to nodes. The components are depicted in Figure 6.1.

The first part of this chapter covers the communication aspects in our simulator. Different components need to be connected with each other in order to enable a communication flow between network nodes. Therefore, we first describe the construction of a network node's protocol stack. A protocol stack is built from algorithms and applications. We show their basic definition, their functionality and the way they are connected with each other to form a stack. Then, we point out how communication between protocol stacks of different nodes takes place. This is done using network interfaces and media. Network interfaces are required to interconnect nodes. The communication in the real-world takes place via air in the wireless case and cables in the wired case. This is modeled by media, which therefore are used to connect network interfaces with each other.

The second part deals with a detailed description of network nodes. Basically, the components described in the first part can be used to perform basic simulations of computer networks. However, nodes can be enhanced to fulfill more complex and realistic simulations of networks than the basic ones. Possible enhancements include the hardware simulation of a specific node's architecture, energy and memory management of nodes, as well as the use of optional hardware devices, like GPS devices. This is described in the second part of this chapter.

6.1. Algorithms

Algorithms are a crucial part of every network simulation. In most cases, algorithms are the thing of interest when simulating a network. Typically, one wants to see how a specific algorithm behaves or performs. Due to this, simulations are made to evaluate and analyze the behavior of algorithms and their interaction.

In Cubus, algorithms can be dynamically loaded and exchanged during runtime. For these purposes an algorithm *base class* provides a basic interface which is common to all algorithms. Besides the adaption of the common interface, a algorithm's internal structure can be freely designed as long as the interface is implemented.

Send- and *receive*-operations belong to this interface, as well as methods for interconnecting algorithms. The send- and receive-operations are used for communication purposes (see Section 6.2.2). Additionally, algorithms have parameters. A minimum requirement is a set of parameters containing a unique identifier, description, type and container of custom parameters. Every component in Cubus is identified by a unique identifier. A textual description is required for logging and other human readable purposes. As we will see, an algorithm's type is important for communication between algorithms. Finally, a parameter container is required to use custom parameters. This is described in Section 6.1.1 in greater detail.

Other network simulators like GTNetS and NS-2 also provide an interface for algorithms. But in most cases, except OMNet++, it is not possible to load algorithms during runtime. Inserting new algorithms enforces a recompilation of the whole simulator system which can be very time consuming.

Besides the load-ability during runtime there is another advantage of our approach. Only those algorithms really required at a specific point of time in the simulation are present in the memory. This keeps the simulator's core small and therefore saves memory.

6.1.1. Custom Parameters

As described in the previous section, an algorithm developer may implement the internal structure of an algorithm in any way, as long he is conform to the interface of the base class. Unfortunately, C++ does not provide *reflection* like Java [McCl98]. This leads to the problem that the simulator is unaware of an object's internal structure during runtime [Nort00]. However, we sometimes need information about an algorithm's internal structure, e.g. if we want to manipulate or access one of its parameters from outside the algorithm. This can only be done if we know what parameters exist. However, in standard C++ we do not have this possibility.

However, some efforts have been made to make reflection available to C++. Three of the most advanced approaches are the *C++ Reflection Package* [Kniz04], an approach using *Metaclasses* [Voll00] and the *Simple Reflection library* [Perl03]. But all of these approaches still are not matured and sometimes are pedestrian. [Kniz04] enhances the C++ *Run Time Type Information* (RTTI) to provide some kind of reflection. The latter one focuses on templates, which may cause some portability problems. [Perl03] uses XML files to describe an object's structure

Cubus follows another way. It is somehow similar to the approach given in [Perl03]. However, our solution does not require XML, but similar configuration files.

We have seen in Section 5.5.1, how configuration files are used to describe custom parameters of algorithms. This information about the parameters is stored in the algorithm repository. To gain access to these parameters, the algorithm's base class defines generic *get*- and *set*-methods. These methods use the container incorporated in each algorithm as a kind of *lookup table*. The container associates an entry, a parameter's description or key, with an action.

For instance, if there is a call to get a parameter P, P is looked up in the container. The associated action returns the current value of P to the caller. By this means, an arbitrary number of user-defined (custom) parameters can be used with each algorithm without required changes to the interface.

6.2. Protocol Stack

The Cubus *protocol stack* is primarily based on the ISO/OSI reference model [Tane03], but is not restricted to it. A protocol stack of Cubus can have an arbitrary number of layers. This enhances the simulator's flexibility. New systems and algorithms can be tested which require different layers. This allows realistic and adaptable modeling of real-world protocol stacks.

A protocol stack is built from its base units, the algorithms. These are interconnected with each other. Each node in the simulation has its own protocol stack.

The next section describes the construction of protocol stacks from algorithms. Then, the communication aspects are discussed, ranging from communication between different layers of the same stack to "inter-stack" communication, i.e. communication between two or more protocol stacks of different network nodes.

6.2.1. Construction of the Protocol Stack

Protocol stacks may consist of an arbitrary number of layers. A restriction to the seven layers of the ISO/OSI model or even five layers in the TCP/IP model would be to grave and could hinder from developing new systems which. New applications or systems may require additional layers, e.g. specific security layers, or need to split existing ones. Using additional layers in other simulators is a difficult task. We have seen in Chapter 3, that protocol stacks in both GTNetS and GloMoSim are restricted to a predefined number of layers. Therefore, we provide an easy to use and extendible design for building protocol stacks.

Each layer in turn may consist of at least one or any number of algorithms. Algorithms of vertically neighboring layers, e.g. layer n has the neighboring layers $n-1$ and $n+1$, are connected in the desired way.

There are two possible ways of connecting algorithms with each other. The first one uses *tree structures* to build a protocol stack. Whereas the second ones connection is based on *graphs*. But only the latter one is able to cover all possible scenarios of interconnection and is therefore used with our simulator. For the sake of completeness and because the tree-based method has some minor advantages, we will outline it in the following. Thereafter, a detailed look is taken at the actual solution, the graph-based connection of algorithms.

Tree-based Connection

Using the tree-based method of connecting algorithms, the protocol stack is built bottom-up, i.e. from the lowest layer (e.g. physical layer) to the top layer (e.g. application layer). Doing so, an algorithm of layer n can be connected to an arbitrary number of algorithms of neighboring upper layer $n+1$. This is depicted in Figure 6.2, where the small boxes in each layer symbolize algorithms.

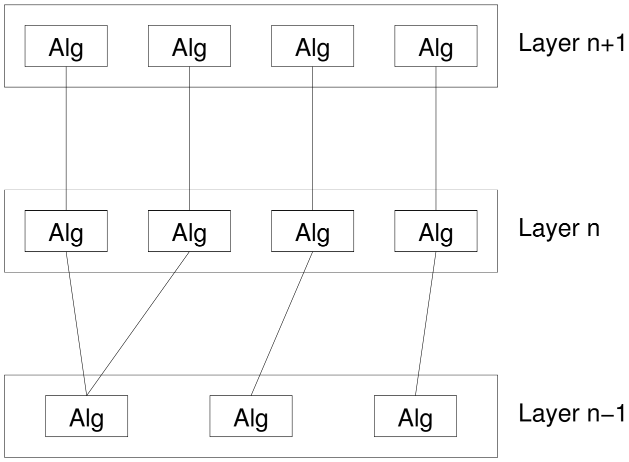

Figure 6.2.: Tree-based connection of algorithms

This approach allows fast stack traversals, especially if the protocol stack is traversed in top-down manner, i.e. starting from the top layer. An algorithm of layer n is all connected to exactly one algorithm of the lower layer $n-1$, i.e. we do not have to determine the correct algorithm at layer n we want to send a packet from.

However, some problems arise when using this approach. Consider the following example. Assume, two algorithms a and b of layer n want to connect to an algorithm c on layer $n+1$. This may destroy the tree property, that trees have to be *cycle-free*, e.g. if an algorithm d on layer $n-1$ is connected to both a and b. Therefore, we have to keep two instances of c, namely c' and c'' on layer $n+1$ to preserve the tree properties.

This approach requires additional memory. Even in this small example to identical copies of the same instance have to be kept in memory. This can exhaust memory usage. To overcome this problem, a new approach is required which is discussed in the following paragraph.

Graph-based Connection

Another way of connecting algorithms is based on graphs. Here, we try to reduce the amount of memory required, rather than keeping multiple instances of the same object in memory, we use only one instance. Thereby, full flexibility is preserved. Our approach is similar to the ones used in BSD's *netgraph* [Cobb00] and the χ-*Kernel* [HuPe91].

To realize that, a graph-like structure is required, as depicted in Figure 6.3. An algorithm A of layer n is connected with an arbitrary number of algorithms of layer $n+1$. Additionally, A can be connected to any number of algorithms of lower layer $n-1$. This approach requires only one instance per protocol stack. The connections between two algorithms are bidirectional, i.e. messages can flow in both directions, upwards and downwards.

This allows to build any dependency structure between algorithms. The structure that results from this way of connecting algorithms represents a graph. Algorithms may be

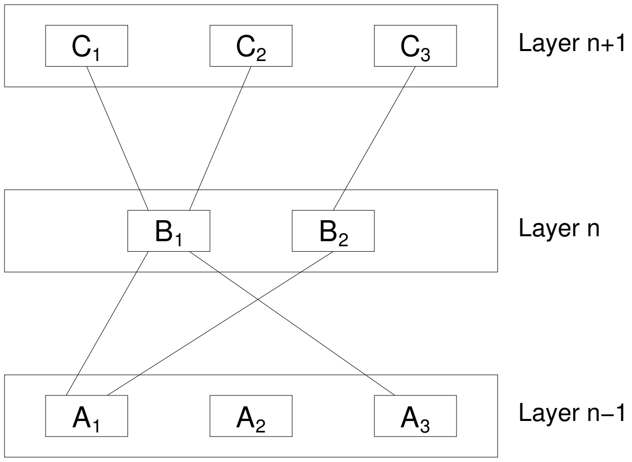

Figure 6.3.: Graph-based connection of algorithms

connected in any way the user wants to. This keeps the stack flexible and adaptable to any purpose.

Each algorithm contains references to its upper and lower neighboring algorithms. These are stored in two separate fields. We refer to them as the *lower field* for the neighbors of the lower layer and *upper field* for the other ones, respectively.

Figure 6.4 shows these two fields. It is a detailed view on algorithm B_1 of Figure 6.3. The upper field of $B1$ denotes algorithms C_1 and C_2 as its upper neighbors. A_1 and A_3 are B_1's lower neighbors.

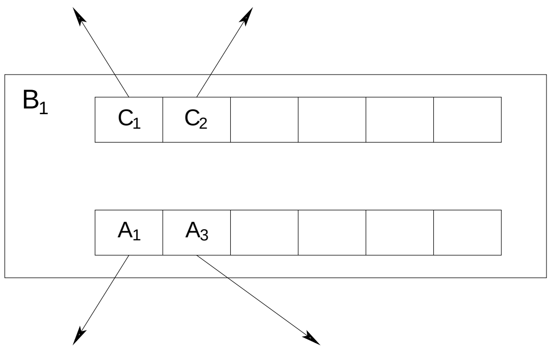

Figure 6.4.: Internal view on algorithm B_1

We have seen in this subsection how algorithms of different layers are interconnected. The next subsection discusses the communication between these algorithms either of the same stack or protocol stacks of different nodes.

6.2.2. Communication between Algorithms

This section discusses the communication between algorithms. We distinguish two different ways of communication:

1. *intra-stack* communication
2. *inter-stack* communication

The first way is concerned with communication between two neighboring layers of the same protocol stack on the same node. The second one describes communication between different protocol stacks and by this between different nodes.

In addition, two types of communication exist. The first one is *event-based communication*, whereas the second type is *method-based*. Both are used in Cubus and are introduced in the following subsections. Afterwards their interaction and combination is examined.

Event-based Communication

Cubus is a simulator based on discrete event simulation. Thus, it uses events as a means of communication. The basic principles of events and their processing were described in Section 5.3.3 in greater detail. Events are generated at specific points of time during simulation, e.g. when a packet is going to be transmitted. After its generation, the event is scheduled in a queue until it is processed.

Here, we apply event-based communication on intra-stack communication. Using this principle, algorithms need not be connected to each other directly, i.e. packets are not passed between neighboring layers directly. In fact, an event is generated whenever a packet is going to be passed to another algorithm of a neighboring layer. This event is immediately scheduled in an event queue. However, this event may be processed at a later point of time, depending to the state of the queue and the event's time-stamp.

Because many packets exist which currently are passing layers at any point of time during the simulation, using the simulation queue (see Section 5.3.3) would clog it in a short period of time. Processing of events is costly. They have to be stored in the queue, taken from the queue. Afterwards they have to be analyzed and proper handlers have to be invoked. This is a time-consuming procedure. The time required for this procedure of course increases with the number of events stored in the queue. Inevitably, this slows down the simulator.

Besides this, clogging seems not to be a serious problem in our case, where the simulation does not run in parallel or distributed, we want to make distribution and parallelization possible at a later point of time. Therefore, assume a distributed or parallel simulation environment in the following.

All the events in the simulation queue have to be managed and processed by the queue. This leads to performance problems if the simulation scenario consists of a huge amount of nodes and algorithms. The simulation queue is a bottleneck in this case.

Another possibility is to use *local queues* at each node. Those are only used for node internal events, e.g. the packet passing events described above. This prevents the bottleneck previously mentioned. But other problems arise. One of these new problems are synchronization problems between local queues on different nodes and between local queues and the simulation queue.

An event causes a change in SimTime. As a result, SimTime needs to be updated very often. This, again, leads to another performance problem. Each event generated for packet passing would need to change the SimTime and therefore would require access to it. Consider a huge amount of nodes where many events are generated. All of them require access to SimTime just for passing a packet from layer n to $n-1$ or vice versa. Additionally, SimTime needs to be Mutex-protected.

Method-based Communication

The second type of communication is based on the invocation of methods. Algorithms of neighboring layers are connected explicitly by using appropriate primitives. This approach of connecting algorithms is similar to the one used by GloMoSim (see Section 3.2.3). Thus,

```
Ln::send(x){
        attach_header(x);
        Ln-1::send(x);
};

Ln-1::receive(x){
        detach_header(x);
        Ln::receive(x);
};
```

Listing 6.1: Send and Receive Primitives

each algorithm has *send* and *receive* primitives used during packet passing. Their basic functionality is depicted in Figure 6.5. A more detailed view is provided in Listing 6.1.

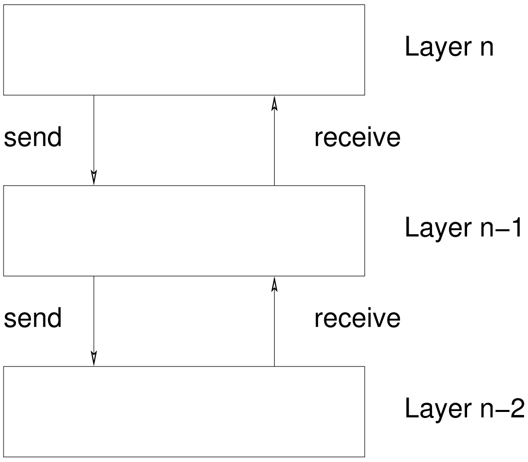

Figure 6.5.: Send and receive between neighboring layers

Assume an algorithm A_1 at layer n wants to send a packet to an algorithm A_2 at the vertically neighboring lower layer (layer $n-1$), i.e. the packet traverses the stack towards the physical medium. Thereby, A_1 attaches its header to the packet and invokes A_2's send primitive. For that it looks up A_2 in its lower field.

Receive works analogous. A_2 at layer $n-1$ detaches its header from the packet. After that it looks into its upper field to find the corresponding algorithm, A_1, to which the packet must be forwarded. Thereafter, A_2 calls A_1's receive operation. By this, the packet is handed over to A_1.

This approach has at least two major advantages compared to the event-based communication principle. First, method-based communication is a bit faster than event-based communication. Because the handling of queues and events in these queues are dropped. A direct method invocation is faster than the whole process of storing events in a queue, taking them from it and handling them with their associated handlers.

Secondly, if we have a look at a potential future parallel or distributed simulation environment, another advantage arises. No access to a central component is required, like to the simulation queue. This prevents a bottleneck in the simulation. Since the nodes work independent from the simulation engine most of the time, they offer good parallelization capabilities.

Recapitulating, one can see, that method-based communication bears some advantages towards the event-based principle. But nevertheless, our simulator is based on discrete event simulation and therefore requires the event-based communication principle. To overcome the disadvantages of this approach, we combine both methods in our simulator. We use method-based communication where possible to enhance the speed of our simulator.

Combination of Both Communication Methods

As described previously, packets are passed inside each node's protocol stack using methods. Communication between nodes, i.e. inter-stack communication, is based on events. This allows for Cubus as a discrete event simulator.

Since we use both types of communication in Cubus, there has to be a transformation from method-based to event-based communication during the data flow at some place in the protocol stack. In fact, the transformation occurs at the transition between the MAC layer and the lowest layer, i.e. the physical layer, more precisely at the *network interfaces*. Network interfaces are introduced in Section 6.4, but for now a coarser definition is sufficient. Network interfaces form the lowest layer of each node's protocol stack. They connect a node, i.e. a node's protocol stack to a medium.

The send primitive of a network interface is invoked by the MAC layer in the same way as between algorithms of other layers. The MAC layer typically forms the vertically neighboring upper layer of algorithms for network interfaces. Inside the network interface, the packet is coded into a `PACKET_TX` event. This event is then handed over to the medium by scheduling it in the simulation queue. Receiving works analogously. As soon as an appropriate event is received by an network interface, e.g. a `PACKET_RX` event, it is decoded into a packet and delivered to the correct upper layer algorithm.

Nodes only need to access one of the simulator's central components, e.g. the simulation queue, at the bottom end of the protocol stack. The node's remaining processing can be done independently from the core engine. This approach enhances performance. Especially, if we consider the parallel and distributed case. But it also allows to use method-based communication in many cases which is in turn used to speed the simulation.

6.2.3. Substitution and Insertion of Algorithms

One of requirements of our simulator is the possibility of replacing algorithms "on the fly", i.e. during runtime, without the need of restarting the simulation or even recompiling the simulator as in the case of GTNetS 3.2.2. Most of the simulators we have examined during this thesis, do not provide this capability. However, restarting a whole simulation can be very time consuming, depending on the simulation's complexity and the time it has already run. In addition, Cubus is supposed to be a combination of a simulator and demonstrator (see Chapter 4). Our design supports the feature of loading and exchanging algorithms during runtime. Especially in the demonstrator case, "on the fly" substitution enhances the flexibility during usage, e.g. one can show influences of specific routing algorithms by replacing one with another.

Before a substitution of algorithms is carried out, the simulation has to be paused. It has to be guaranteed that the algorithm that is going to be replaced is not *active*, i.e. it is not processing any packets at the moment of substitution. After pausing the simulation, the corresponding upper and lower fields in the specified protocol stack have to be updated. Thereby, the connections of the old algorithm have to be maintained. The upper and lower field of the new algorithm are updated according to the old algorithm's fields. Additionally, the lower field of the algorithms of the directly neighboring upper layer that have been

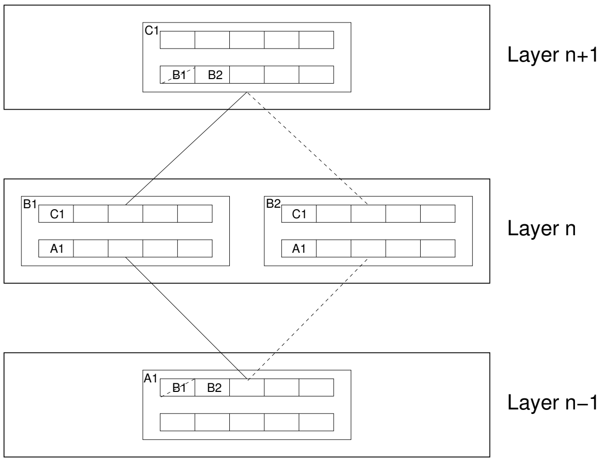

Figure 6.6.: Algorithm B_2 replaces B_1

connected with the old algorithm are updated. The same is done to the upper field of the neighboring lower layer's algorithms. Now, the old algorithm is replaced by the new one and the simulation can be resumed.

A simple substitution is depicted in Figure 6.6, where an algorithm B_1 of layer n is replaced by algorithm B_2. The dashed lines show the changes *after* the substitution. To simplify matters, the old values of the lower and upper fields are kept in the figure.

However, insertion is much simpler. A new algorithm is created and connected to the desired upper and lower layer algorithms. The simulator does not need to be paused during this connection. Since this algorithm is unknown and should not be used before it its "installed" completely, there is no way this algorithm can get active accidental.

But there is a potential problem when replacing algorithms. The new, altered stack may no longer accept packets, it has done before. That's due to the fact, that if the packet has a header for B_1, A_1 does not find that algorithm anymore. Therefore it will discard the packet.

6.3. Applications

The ISO/OSI model introduces the *application layer* as the topmost layer of each protocol stack. Among other things it is used to do some some semantic conversions between associated application processes and it controls the basic data flow in the network.

We use *applications* to model this behavior. Applications are a special and extended variant of algorithms. As in the ISO/OSI model they form the topmost layer of the protocol stack.

6.3.1. Applications in Existing Simulators

Other simulators provide application-like components which either roughly model the ISO/OSI model's application layer or are used to introduce some kind of endpoints at the protocol stack that extend the simulator's capabilities.

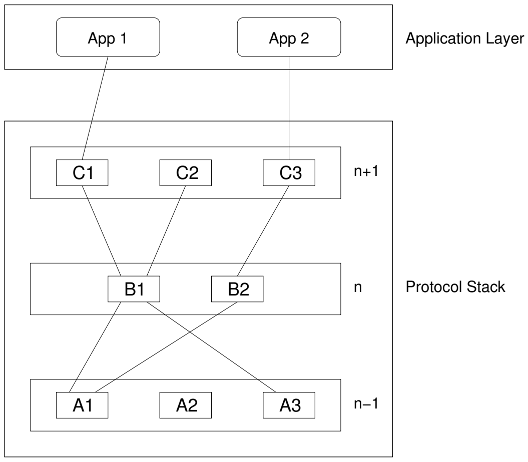

Figure 6.7.: Protocol stack extended by applications

NS-2

The equivalent of applications in NS-2 are the *application objects* ([FaVVp04], Chapter 35). *Agents* ([FaVVp04], Chapter 10) are used as an interface between the transport layer of the protocol stack and applications. Agents represent the endpoints of a protocol stack, e.g. packets of the network layer of a protocol stack are generated and consumed there. Therefore, they support packet generation and reception.

Agents can be connected to the application objects of NS-2. Two basic types of applications are distinguished in NS-2, *traffic generators* and *simulated applications.*

GTNetS

In GTNetS applications describe the flow of data through a defined network topology. They create data demands on the network according to the application's behavior. A great variety of applications exist in GTNetS, like CBR applications that generate constant bit rate data between network nodes. Applications are attached to the top layer of the protocol stack. Thereby, they control the packet flow.

GloMoSim

GloMoSim supports applications, too. In GloMoSim applications form the application layer and are connected to the transport layer via an appropriate interface. The applications provided by this simulator are mainly traffic generators for FTP, HTTP, Telnet and CBR.

6.3.2. Cubus Applications

Our applications are extensions of algorithms. Their primary focus is on initiating the dispatching of events. The applications of Cubus can roughly be compared to the traffic generators of NS-2 or GTNetS.

They are extended to directly cope with events. Since "normal" algorithms can also be used for handling events, the distinction between applications and algorithms is primarily conceptual.

Applications can react to events originating from the core engine by default. This predestines them as starting points for a simulation. In addition, they can be used as traffic generators after the simulation is initiated by a core engine event. Consider a situation where the scenario configuration file determines that a specific application of a specific node should broadcast packets in regular periods starting from a specific point of time.

6.4. Network Interfaces

Network interfaces are needed to connect nodes to a specific medium. This section describes our design of network interfaces. First, we examine other simulator's equivalents to our network interfaces. Then, we provide a detailed view on our approach.

6.4.1. Network Interfaces in Existing Simulators

This subsection gives a short overview of type of network interfaces provided by other simulators. We will primarily focus on NS-2 and GTNetS. But a glimpse is taken at GloMoSim, too. All of them provide some basic network interfaces which are used for linking nodes and media.

However, their abilities and tasks differ in some part. Signal propagation is modeled in some simulators within network interfaces, e.g. NS-2, while other provide this at different places, e.g. GTNetS.

NS-2

The standard version of NS-2 [FaVVp04] has only limited capabilities for using network interfaces. However, the CMU Monarch Project [CMU 98] has extended this standard version to provide a more realistic view of network interfaces. The authors of [Bind04] extend the CMU add-on by support of multiple Interfaces per mobile node.

The network interface in NS-2 is responsible for collision detection and provides the possibility to attach a radio propagation model. Two radio propagation models exist in NS-2, *free space path loss* and *two ray ground*. The network interface connects a simulation node to the physical medium. A network interface tags each transmitted packet with *meta data*, e.g. wavelength and transmission power. This information is then used by the receiving Interface to determine if the packet can be received correctly, i.e. if the signal is strong enough (see Section 2.2.2). The network interfaces provided by NS-2 basically represent the DSSS radio interfaces [Schi00].

GTNetS

The equivalent to our network interfaces are called Interfaces in GTNetS. They are used to connect a node with a medium, too. Interfaces simulate the behavior of real hardware interfaces (e.g. network cards). However, nodes in GTNetS can have more than one Interface attached to them innately.

Interfaces encapsulate some information like IP address, MAC address, as well as an address mask. Many links, which are connections between Interfaces, create the Interfaces automatically. So the user is not required to do it himself.

However, other than in NS-2, collision detection and a radio propagation model are not part of the Interfaces provided by GTNetS. Both a rudimentary collision detection scheme and radio propagation model are implemented in the media, e.g. a wireless medium.

GloMoSim

GloMoSim [ZeBG98, BTAT+99] and its commercial version QualNet [Scal04] do not model an exact equivalence to our network interfaces. They can roughly be compared with the GloMoSim *radio models* which encapsulate characteristics of the physical layer and medium and provide an antenna model. In addition, separate propagation models are used which, roughly spoken, belong to the medium.

6.4.2. Cubus Network Interfaces

Network interfaces interconnect nodes with different media via which communication takes place. A description of *medium* is given in Section 6.5. This means that without network interfaces there is no way for nodes to communicate with each other.

In Cubus, each node may possess an arbitrary number of network interfaces and can therefore be attached to one or more media. This makes it possible for a node to communicate via these different media, e.g. it may be attached to a wireless and a wired medium. Using two interfaces to connect a node to a wireless and a wired network, one can model special hardware like access points for wireless nodes.

Network interfaces can be added to nodes or replace another network interface during runtime. The basic principle of substitution and insertion is analogue to the substitution of algorithms described in Section 6.2.3. New network interfaces can be imported to the simulator during runtime, since they can be dynamically loadable in the same way algorithms are.

Network interfaces represent the physical layer of the ISO/OSI model. They may model parts of the MAC layer as well, e.g. the PMD sub-layer of IEEE 802.11 (see Section 2.3). In doing so, network interfaces encapsulate important hardware properties, which are used to simulate real network adaptors like Ethernet network cards. Antenna models can be defined and incorporated in these network interfaces.

One of the information given in network interfaces is the descriptions of transmission and reception signal strength. This data is required by *propagation models*. A detailed description and design of these models is beyond the scope of this thesis and is topic of another diploma thesis, which is currently in progress. We will come back to propagation models when covering media in Section 6.5.

In few words, as already illustrated in Section 2.2.2, signals are transmitted with a specific signal strength, the so-called *transmission signal strength*. The signal weakens during its propagation through the medium. If the signal's strength is still above a well defined threshold at reception, the signal can be received correctly. This threshold can be calculated by using the *reception signal strength*.

The next subsection describes how network interfaces can be used with other simulation components, i.e. how they interact with them. A main point of interest is the connection between the lower layer algorithms of a protocol stack and a medium.

Connection to other Components

Network interfaces need to be connected to algorithms of a protocol stack and to media. In Cubus, network interfaces are treated as a special kind of algorithm (similar to applications). This eases the interconnection between protocol stack and network interfaces.

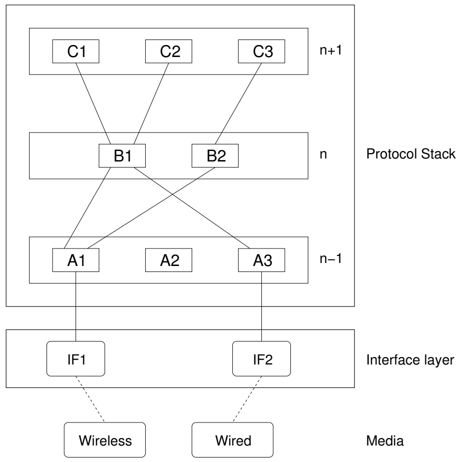

Figure 6.8.: Interconnection of network interfaces

Network interfaces are regarded as the lowest layer of the protocol stack. Therefore, they are connected to algorithms in almost the same way as “normal” algorithms are interconnected with each other. The same procedure as in Section 6.2.1 applies, except for the lower field. Since network interfaces arrange the transition from method-based to event-based communication, they are only connected to upper layer algorithms. Considering the connection to the medium, there is no need for a connection like the one described above. Media receive packets encoded into events. The interconnection between medium and network interfaces, is not as general as the interconnection between algorithms. It is a one-to-one connection, i.e. one network interface is connected to exactly one medium. The basic connection between a protocol stack and network interface is depicted in Figure 6.8.

Processing Packets and Events

After network interfaces have extended an already existing protocol stack by being attached to it, packets can be transmitted via inter-stack communication. As an example, consider the case where a packet is processed at a MAC layer algorithm of a protocol stack. Typically, the MAC layer is connected to the physical layer and thereby to a physical medium. This is modeled as a network interface in our case.

As soon as processing of the packet is finished at the MAC layer, the packet is handed over to an appropriate network interface by invoking its send primitive. This is done in the same way as in the standard intra-stack communication (see Section 6.2.2). By this means, the packet is made available to the network interface.

As outlined in Section 6.2.2, the transition from method-based to event-based communication is done in the network interface. Thus, the packet is encoded into an event and scheduled at the simulation queue for further processing in the medium. In contrast to NS-2, the event and the packet need not be tagged by meta data. Moreover, calculations for signal loss and appropriate thresholds are carried out by using the network interfaces involved in the current communication. This is done based on propagation models.

If an network interface reacts to an event, e.g. if it receives a packet, it has to process it. During this processing he event is decoded into a packet, the packet that actually was transmitted. Following, the appropriate algorithm in the upper field is determined and the packet is passed to that algorithm by invoking its receive principle.

Using this principle we are able to transmit a packet from one network interface to other ones. As soon as the packet leaves the network interface, it comes into the authority of a medium which in turn applies further processing. To close the gap in inter-stack communication, we describe media in the following section.

6.5. Medium

A *medium* is used to connect nodes with each other, i.e. the inter-stack communication is realized via media. A medium implements parts of the physical layer. It encapsulates physical characteristics of the real world medium. Before we describe our approach in greater detail, we glance at media or media equivalents of simulators other than Cubus first.

6.5.1. Media in Existing Simulators

The Cubus medium is related to *links* in other simulators. In NS-2, there exist *point to point links* connecting two nodes with each other. That represents a physical wire connection. In the wireless case, connections are realized using *network interfaces* (see Section 6.4). GTNetS however uses links in both cases, wired and wireless. Therefore some basic types of links exist, ranging from *wired links* to *wireless links.* These encapsulate characteristic physical properties of each link type.

6.5.2. Cubus Medium

Our simulator follows a similar principle. The medium is responsible for the treatment of signals, i.e. their propagation, transmission errors, and so on. Using other components like the *Environmental Model* and the *Location Service*, relationships between nodes, or more precisely network interfaces, are determined and used.

One can add an arbitrary *error model* to a medium to simulate packet loss and transmission failures. Error models are described in Section 6.5.4 in greater detail. But at first, we describe the interaction between a medium and other components of the simulator.

6.5.3. Connection with other Components

As already mentioned, nodes are connected via a medium. More precisely spoken, the connection is between the network interfaces of nodes, e.g. two nodes are connected via their wireless network interfaces through a wireless medium.

The communication process, the inter-stack communication, is based solely on events. A node intending to transmit a packet schedules a specific event at the simulation queue. Then, the event is processed by taking it from the queue and handing it over to the appropriate medium.

Whenever the packet is received by the medium, it is analyzed in order to determine the source network interface, i.e. the network interface it was sent from. After that, all network interfaces in detection range are determined, i.e. all network interfaces that receive the packet either correctly or are able to recognize that a transmission has occurred. The interference range is neglected, since it causes complex physical calculations, e.g. differential equations, and in most cases, their contribution to simulation accuracy is small.

Thereafter, from all network interfaces those are selected that are in transmission range, i.e. those that are potentially able to receive the signal correctly.

The packets for these network interfaces are guided through an associated error model which marks packets as erroneous according to a specific probability distribution. This is used to simulate the medium's unreliability. Finally, a corresponding event is scheduled, e.g. `PACKET_RX` or `PACKET_FAIL`. The receiving network interface uses these events to determine if the transmission is successful or if an error occurred.

Each medium has an associated propagation model. It can be replaced by another model to simulate different behavior, e.g. *free space path loss* replaced by *two ray ground* [GrKL04]. We use propagation models to determine those network interfaces that are within transmission range, e.g. free space path loss (see Section 2.2.2).

6.5.4. Medium Error Model

Real-world media, like wires and air, are unreliable by nature, whereas the error rate in air is much higher than in wires. Error rates in wires are significantly low.

However, at first, a typical simulated medium is ideal, i.e. error-free. To overcome this gap, many simulators employ error models with their media to simulate packet loss ([FaVVp04] Chapter 13). Cubus also uses *error models* to simulate packet loss.

Error models can range from very simple ones to complex models [Lemm02]. However, with increasing complexity, the accordance with reality increases, too.

Cubus uses error models that can be attached and removed to a medium. Their internal structure may be arbitrary but will typically follows some sort of probability distribution, e.g. Gaussian, Pareto. These models are employed as follows.

The propagation model hands over all network interfaces that potentially receive the packet successfully. The error model determines a subset of these network interfaces where the packet is to be lost, i.e. its transmission has some error. According to this corresponding events are generated as described in the previous subsection.

6.6. Nodes

The previous sections have described the communication process within our simulator. We have glanced at intra- and inter-stack communication. We have seen how a custom protocol stack is built, how packets can be generated, e.g. in applications and how these packets are spread among network interfaces. Network interfaces belong to nodes. Each node may have one or more of these interfaces. A complete protocol stack is incorporated inside a node. In the following sections, we discuss the design of network nodes and several possibilities to enhance them in order to get a custom defined level of realism.

Nodes are an integral component of every simulator. They can be considered as computing units on which algorithms and other components run. These computing units range from very small sensornet nodes to workstation PCs and even servers. The attachment of different network interfaces allows to build special hardware devices like access points for wireless networks.

Some simulators differentiate between mobile nodes and stationary nodes. The former has properties that go beyond those of fixed nodes: First, mobile nodes may move. Therefore, locations are more important for mobile nodes than for stationary ones, since they may change their position continuously. Secondly, mobile nodes usually have limited resources, ranging from small memory to a limited energy source like a battery.

These properties have to be modeled for mobile nodes. That increases the complexity of nodes, since a great amount of new dependencies arise. For example, sending and receiving only works if enough memory and energy is available.

In Section 6.6.2 we discuss the basic configuration of our simulator's nodes, describing the components that can be attached to them and the features they provide. In the following we briefly show how nodes can be extended by different components. Finally, Section 6.6.3 describes the simulation of a real node's hardware that is incorporated by our approach. But before we discuss our approach we examine nodes of simulator's other than Cubus.

6.6.1. Nodes in Existing Simulators

As we have seen, network nodes are an integral part of every network simulator. Therefore, we outline features of network nodes of other simulators in this subsection. We thereby focus on properties that are relevant to our simulator and discuss some deficiencies of these simulators' nodes.

NS-2

The standard distribution of NS-2 only supports stationary nodes. However, the Monarch Project [CMU 98] extends nodes to incorporate characteristics of mobile nodes. Multiple agents can be attached to each node which influence the simulation, as outlined in Section 6.3.1. In addition, each node manages a list of available neighbors. The information about neighbors is required for communication. For more detailed information about nodes in NS-2, see [FaVVp04] Chapters 5 and 16.

However, nodes as provided by NS-2 have some deficiencies. As we will see in Sections 6.8 and 6.9, they lack proper energy management as well as memory management. Additionally, configurations of nodes in NS-2 are static, i.e. changes during runtime are hard to realize. Extensions to nodes require a recompilation of NS-2. Another problem is that only one network interface can be attached to a mobile node in the Monarch adaptation. The restriction to one network interface prevents modeling of all possible real-world nodes.

GTNetS

Nodes provided by GTNetS have a number of objects associated. Among these objects is a mobility model for mobile nodes, and one or more Interfaces. As described in Section 6.4 these interfaces mediate between the MAC layer and a communication link. During the creation of a node it is assigned a routing object as well as IPv4 automatically. After the creation, interfaces and links can be added to a node. This describes the connectivity of a node.

However, as in NS-2, nodes in GTNetS are static. Altering a nodes structure, including interfaces and algorithms, during runtime is not possible. Extensions require a recompilation of the GTNetS library. Energy management of nodes in GTNetS is not flexible. It is a fixed, hard-coded battery model that this implemented with GTNetS. A new model requires implementation changes within the base classes.

GloMoSim

Nodes in GloMoSim differ from the design of nodes we have previously seen in NS-2 and GTNetS. Here, node aggregation is used to reduce memory usage and enhance the performance in a parallel simulation environment. Therefore, the network is divided into several partitions. The network nodes in one partition are combined in an aggregation. They typically react in the same way to events. Only limited enhancements are individual to each node, e.g. an energy model.

This approach speeds up a simulation dramatically, as outlined by the authors of GloMoSim. Unfortunately, our simulator requires individual instances of each node to access its characteristics and components, e.g. each node may have a unique protocol stack or its own battery model. Therefore, the GloMoSim approach may only hardly be adapted for our simulator.

6.6.2. Basic Node Configuration

As we have seen during our examination of other simulators, nodes and their configuration are static in their approaches. Our simulator and demonstrator requires more flexibility and adaptability, especially due to its purpose as demonstrator. We want to be able to extend nodes by other components to fit our needs depending on the actual simulation we are doing.

In Cubus, there is no distinction between *mobile* and *stationary nodes*. The latter are considered as a particular case of mobile ones with restricted properties. They are restricted in the sense of lacking mobility and having potentially unlimited energy resources.

Nodes are characterized by a number of properties. Among them are an unique identifier and some descriptive parameters like in other Cubus objects. Additionally, there is information about the node's hardware architecture that is going to be simulated. Important parameters for this kind of simulation are the number of bits of the node's architecture and the BogoMips [vDor04] of this system. Section 6.6.3 describes in detail how this hardware simulation is realized.

Furthermore, nodes can be enhanced by attaching other components. The possible components are depicted in Figure 6.1. Some of them are required for correct operation. One of the most important components is the protocol stack which is unique for each node. Each algorithm instance in the stack is specific to a node. This is necessary to allow runtime changes at the stack of a particular node. The stack is constructed in the way presented in Section 6.2.

The stack is connected to network interfaces that are linked to a node. The network interfaces connect the protocol stack to a medium and by doing this a node to a medium. Therefore, nodes are able to communicate with each other by using this principle.

In addition to an arbitrary number of network interfaces that can be connected to a node, the same principle holds for hardware devices, especially positioning devices. While network interfaces are required for inter-stack communication, positioning devices are only needed by mobile nodes.

For a realistic simulation of nodes, appropriate battery and memory models are required. This is realized by using *Energy Management Units* (EMU) and *Memory Management Units* (MMU), which are described in detail in Section 6.8 and 6.9, respectively.

6.6.3. Hardware Simulation

Simulation of a node's hardware is important, since the hardware architecture of a node needs not be equal to the architecture the simulator runs on. As a rule, they are typically different. They differ, e.g., in the format of address, e.g. 8 bit or 32 bit, and in processing speed.

The following example illustrates why a simulation of the target's hardware architecture is necessary to obtain more realistic results than we get without a simulation like that. Imagine a typical sensornet node with an 8 bit architecture and a CPU of 20 MHz [TecO04]. The simulator runs on a 32 bit processor with 3 GHz. A packet is processed at the node, either when received or before being sent. It is obvious that the real node requires a longer period of time for processing than the simulated node. Therefore, the assumed real processing time and SimTime drift apart. The adjustment of these times is realized by the hardware simulation described in this section.

Basic Principle of Adjusting Simulation Time

The determination of the correct processing duration in SimTime is achieved by consulting the architecture's BogoMips [vDor04] and the real time of processing at the simulated node, i.e. the Wall Clock Time of the processing procedure on the node.

BogoMips are a way of measuring the speed of a CPU. They were invented by Linus Torvalds. At boot time the Linux kernel measures how fast a certain kind of busy loop runs on a computer [vDor04]. Using this information enables the calculation of the processing time in SimTime.

Let t_S and t_E be the times, in Wall Clock Time, when the processing starts, and ends, respectively. Then the duration of processing d_{sim} is given by

$$d_{sim} = (t_E - t_S) \cdot \frac{B_{real}}{B_{sim}} \tag{6.1}$$

where B_{real} denotes the BogoMips of the computer the simulator runs on and B_{sim} those of the simulated node.

The determination follows the principle outlined below. As soon as it is discovered that a packet has reached its endpoint in the protocol stack, e.g. the last algorithm on its way up or an network interface on its way down, an event is scheduled that determines the time of final receive. At this point in time the treatment looks like the one shown in Listing 6.2.

Assume, a packet is received by a node. Then, the packet goes up the protocol stack. Thus, the arrival time is noted, both in SimTime and in the actual WCT. The SimTime is given by the event, `PACKET_RX`. Now the packet traverses the stack upwards, either until no corresponding upper layer algorithm is found or the packet is received by the top layer algorithm for this packet. At that time the duration of processing is determined by using

```
Δt = t_E − t_S;
d_sim = Δt * (B_real / B_sim);

if (currentSimTime < simTime_start + d_sim){
        create UpdateEvent e;
        e -> setTime(simTime_start + d_sim);
        schedule(e);
} else {
        ignore;
}
```

Listing 6.2: Calculation of processing time in SimTime

Equation 6.1. Thereby, a specific update event is set off which is required to update the SimTime. But that is only necessary, if the SimTime has not already progressed to later point of time. This event is scheduled in the simulation queue and defines the time of final receive of the packet in the node. Until this event is processed other events may occur in the node, e.g. other packets may be received and processed by the node. This roughly models the parallel work on real hardware.

The other way round follows the same principle. The update event is generated when sending a packet to the medium, in most cases this is the packet transmitting event itself. The beginning of processing happens either at the application layer (see Section 6.3) or at a specific algorithm in the stack.

Problems with Hardware Simulation

As we have seen in the previous paragraph, a simple hardware simulation can be achieved by following the principle outlined there. But there is a problem with this approach, we want to discuss in this paragraph.

To understand the problem we first have to examine the functionality of real-world network nodes. We only consider a simple case where one network interface, e.g. a Ethernet network card, is attached to the node.

As soon as a packet is received by the network interface of a node, it is processed by going up the protocol stack. But during this time of processing the packet new packets may arrive at the network interface or packets are sent from an application to the network interface. Since real-world nodes support parallelism, this is no problem as long as different resources are used.

This type of parallelism cannot be simulated in a sequential simulator. Our approach tries to model this behavior by scheduling an event, e.g. for a final receive. The SimTime is calculated in the way described in Listing 6.2. Let us assume that t_{final} is the adjusted SimTime of the event, while $t_{current}$ is the current SimTime of the simulator. If there are any events in the simulation queue whose time-stamp is smaller than t_{final}, they still can be processed. There may be events for the same node having scheduled this final receive event. Thereby, the node may still process packets that arrive in the period of time between $t_{current}$ and t_{final}. This roughly models the real-world behavior but still provides some problems. First, our approach does not care about using the same resources on the node, i.e. packets may be processed by the same algorithm at potentially the same time. Secondly, specific custom values in algorithms or the node may have been changed in the simulator, although the packet is not yet received completely.

Besides these problems our approach allows simulation which is a little more realistic than simulations without coping with hardware aspects of the network nodes.

6.7. Hardware Devices

We have seen in Section 6.6.1 that nodes in simulators other than Cubus are static. Extensions require a recompilation of the simulator in most cases. But extensions to nodes may be necessary, e.g. a mobile node requires some kind of positioning system. Additionally, nodes may have attached one or more devices. Our approach of using *hardware devices* allows easy and flexible extensibility of nodes. If dynamic loading of hardware devices is supported during runtime, we do not need to recompile the simulator each time a change to the node's configuration occurred.

Hardware devices are components that can be attached to a node to enhance its functionality. NS-2, GloMoSim, GTNetS and all other simulators that were examined during this thesis do not provide a similar possibility, i.e. none of these offer the potentiality of enhancing nodes in a simple and easy way.

Hardware devices form a base class for many types of extensions. Therefore, they only require little information. Every hardware devices has to have a unique identifier, like all Cubus objects, a description and a type. The type is used to classify hardware devices, e.g. positioning devices like GPS-receivers, infrared-receivers.

Positioning devices are the most important representatives of hardware devices. They are discussed in the following.

6.7.1. Positioning Devices

Real-world mobile network nodes require some kind of positioning device. In most cases these are GPS-receivers nowadays. They are used to determine a nodes position and influence its behavior thereby. We use *positioning devices* in our approach to model the behavior of such real-world devices.

Positioning devices comprise a great variety of possible devices. We identify two categories of positioning devices. On the one hand those that work with actual *geographical coordinates*. On the other hand devices that use *symbolic coordinates*. GPS-receivers are an example for the first category. Infrared-receivers belong to the latter one.

Before introducing the design of positioning devices of Cubus and how we attach them to nodes in detail, it is reasonable to examine other simulators. The question is how they handle positions of nodes. However, we will not discuss how positions are calculated or assigned to nodes. This is beyond the scope of this thesis. Primarily, we focus on position management inside of nodes.

Positioning in Existing Simulators

In GTNetS positions are directly stored in each node. They are restricted to exact, physical ("real world") position. The x- and y-coordinates of nodes are given in meters. Each node may only have one position, since it only includes one pair of values (x, y). Inaccuracies and falsifications of positions cannot be realize directly using nodes.

NS-2 handles positions in almost the same way ([FaVVp04] Section 16.1.2). In addition to GTNetS, NS-2 theoretically supports three-dimensional environments. At least it makes a third position variable, z, available. But this coordinate is always forced to be zero in the current version (version 2.27). Both simulators update the coordinates while the node moves.

The approach chosen by OMNeT++ [Varg01] is a bit different. Positions of nodes are directly associated with their mobility models. Each node has its own instance of a mobility model and its current position is stored within it.

A more detailed analysis of positioning principles is part of another diploma thesis and therefore is beyond the scope of this thesis.

Cubus Positioning Devices

Cubus' position handling in nodes follows a much more flexible approach. While being easy and straightforward, there is one major drawback with the way the other simulators realize position handling. Their approach does not accurately model reality when they assign positions directly to a node.

Today's mobile nodes may incorporate more than one positioning device, e.g. a GPS- and an infrared-receiver. Additionally, differences in their accuracy is possible, e.g. two different GPS-receivers may differ in the positions they compute depending on their antennas. Simulators other than Cubus need to be modified to handle this feature, since they only store the node's exact positions.

Cubus encapsulates the properties of real hardware positioning devices in its own positioning devices. These devices can be attached to nodes arbitrarily.

The positioning device provides one common interface. It allows reading and writing of positions among other things, e.g. influencing a node's energy level by consuming energy. They are associated with the *location service* which makes the positions of all nodes available. This service is part of another thesis.

A device queries the location service in regular time intervals. The location service then returns the current exact position of the node this device is attached to. This position is stored in the device and can be further processed. A GPS-receiver may compute inaccuracies and produce imprecise locations. This can be used to simulate the influence of satellites or other hardware characteristics.

By using this approach the management of positions in Cubus is simple and elegant. Nodes do not need any information about their positions. Devices can be replaced during runtime, similar to algorithms and network interfaces.

6.8. Energy Management

As we have seen, it is very important to provide an accurate platform for the simulation of algorithms and whole networks, since the testing in a real environment is expensive and time consuming (see Section 3.1.1).

Wired network nodes are usually attached to the standard power line and therefore do not have to cope with energy limitations. But mobile nodes are usually equipped with batteries and therefore heavily depend on their batteries' performance. Because batteries discharge during their lifetime, an accurate simulation environment has to model this behavior.

This section describes the modeling and management of energy in our simulator. To understand the basic problem that arises with the use of batteries, we discuss a batteries' behavior in the real-world in the following subsection. After that, we examine other simulators in order to identify their energy management capabilities. Finally, we describe the approach Cubus provides.

6.8.1. Batteries in the Real World and in Simulation

Before looking at specific implementations and designs of energy models and their management, we have to examine how a battery discharges in the real world. In opposition to the common opinion, batteries discharge in a non-linear way and not in a linear one. A linear discharge would be a linear decrease in available energy. But the non-linear discharge of batteries complicates the required model if it is supposed to be close to reality. Additionally, a battery discharges continuously during its lifetime. It doesn't matter if there is any load on the battery or not. Furthermore, in *idle state*, i.e. no load is on the battery, a battery may partially recover. This is called the *Recovery Effect*. This makes a simulation even harder and complex.

Considering load, each operation on a network node reduces the node's current battery level. That means, there are two phenomena that overlap: the continuous discharge specific to each battery type and the use of energy at each operation carried out by a node.

The way a battery discharges is a very complex electro-chemical chain of reaction processes, i.e. this leads to an exact and almost 100% realistic model, like it was realized in [DoFN93]. Their solution is computationally intensive. Therefore, it should not be used in a simulation environment. The authors of [HaTi03] provide a much more efficient solution with an acceptable low error rate ($< 3\%$).

Both the continuous discharge and the operation level consumption can be simulated in a fine-grained and realistic way or coarser-grained way. The latter inevitably results in some inaccuracies between a real-world battery and the simulated battery.

Fine-grained simulation requires the examination of each instruction carried out on a node. Since each operation requires energy and thereby reduces the node's battery level. This leads to a dramatic computational overhead and therefore slows down the simulation. But a coarse-grained simulation may result in greater error rates which falsify the simulation results. Therefore, a good way in between has to be found and implemented.

6.8.2. Energy Management and Models in Existing Simulators

Since most simulators available are designed for wired networks, they do not provide sophisticated energy models (if at all). This section discusses some simulators which are capable of simulating mobile networks and provide the means for using more or less accurate battery models.

NS-2

NS-2 provides a very simple energy model, which was extended by [HaTi03]. The basic model consists of three values.

- `initialenergy_`
  the initial value of energy available at start time

- `energy_`
  the node's current level of energy
- parameter that decreases the battery level during runtime

NS-2 uses a simplification for its energy model. It uses an ideal energy source with no side effects. For each packet that is received or sent, NS-2 reduces the current battery level by a certain value. Depending on the implementation and version of NS-2 this value is around $30mW$. As soon as the energy level is equal to or smaller zero, the node is considered to be dead. This simplified model allows the demonstration of the basic features of a battery but is far from being a realistic model.

In addition to these features, an *idle energy consumption* can be specified. This discharges the battery when the node is not active (*idle state*).

The model provided by NS-2 assumes a linear discharge of a battery. But only for an ideal energy source this assumption holds. This means that the basic energy model that is provided with the standard NS-2 version is too inaccurate for more realistic simulations. One advantage of this solution is that it is computationally inexpensive. No complex computation have to be carried out in order to discharge a battery. Simple subtractions are sufficient for this model.

The authors of [HaTi03] extend the standard NS-2 battery model by providing the possibility to attach a battery model to a node in addition to the standard energy model. This battery model extends the functionality of the standard model and allows high-level simulation of accurate lithium-ion batteries. This approach supports the recovery effect and the non-linear discharge of batteries among other things.

GTNetS

GTNetS [Rile03a], was specially designed for wireless networks. Version 0.2 introduces a simple battery model for mobile nodes that resembles that of GloMoSim.

A battery is assigned to each mobile node. The battery is represented by a single number that indicates the available amount of energy left. The energy is decreased each time a packet is transmitted. This is done according to the following equation.

$$E_{new} = E_{current} - (\mathrm{Duration}_{TX} \cdot \mathrm{Power}_{TX} + \Delta\mathrm{time} \cdot \mathrm{CPUpower}) \tag{6.2}$$

Whereas $E_{current}$ denotes the current battery level at the time of transmitting, $Duration_{TX}$ the time needed to transmit the packet and $Power_{TX}$ the energy required to transmit a packet unit. In addition to that, $\Delta time$ describes the time between the last battery update and the current time. $CPUpower$ is the CPU power consumption at each point of time.

This simple energy model is used in each of the Interfaces of a node. It roughly resembles a real battery and is far more realistic than the standard NS-2 energy model.

GloMoSim

In GloMoSim and its commercial successor *QualNet* [Scal04] the energy model is implemented in the physical layer. There are four states in this model: *idle*, *sensing*, *receiving* and *transmitting*. However, there is no sleep mode in GloMoSim, which is one of its major drawbacks.

Receiving a packet requires a specific amount of energy. The default value is $900mW$. The calculations for transmissions are more complex than for receiving. The energy consumed during a transmission process is given by:

$$E = (\text{PowerCoef} \cdot \text{Power} + \text{PowerOffset}) \cdot \text{duration} \tag{6.3}$$

where the values of *PowerCoef* and *PowerOffset* are defined in the WaveLAN specification [Luce04]. They are set to $\frac{16}{sec}$ and $900mW$, respectively. *Power* is proportional to the distance the signal has to travel.

A major drawback of GloMoSim's energy model is the lack of runtime support. For each packet transmitted, the consumed energy is calculated and added to the consumption statistic. This value is only given at the end of the simulation. There is no way to get access to it during runtime. Therefore, during runtime every node has infinite energy and will never stop working due to a low battery level.

6.8.3. Cubus Energy Management

The energy model designed for Cubus is somehow more complex than in other simulators like NS-2, since a higher level of realism should be possible. Not all calculations can be done at the node itself, since several independent devices are attached to each node (e.g. hardware devices, network devices). Each device consumes energy, either in sleep mode or by performing operations. Therefore, all attached components must provide their own energy consumption which then has to be combined at the node to get the overall energy consumption. Each devices energy consumption may be calculated as proposed in [MaOb04].

Cubus uses *Energy Management Units* (or *EMUs*) to handle batteries and energy in general.

Each node contains two basic parameters, regarding its energy household.

- `energy`
  the node's currently available amount of energy
- `maxEnergy`
  the node's initial amount of energy. It is mainly used for statistical reasons.

Additionally, each node has a *single* EMU attached. This situation is depicted in Figure 6.9. We see the *energy* parameter which is correlated with an EMU. An EMU is responsible for simulating a real-world energy source. Other than in the NS-2 extension [HaTi03], the source is not restricted to a model of lithium-ion batteries. Every kind of energy source can be modeled.

This plugability allows energy source switching "on the fly". Consider a case where a notebook is modeled. First, it runs on battery. Then, when battery is low, it can be switched to main power supply system by removing the old EMU and replacing it by a new one.

The basic functionality of each EMU is to modify the node's energy level by using the appropriate methods. Two basic cases for EMU's can be distinguished, EMU's for wired networks and EMU's for mobile and wireless networks.

Nodes in wired networks are typically plugged to the main supply. At first glance, this renders an EMU for the wired case unnecessary, since the energy level of a node does not

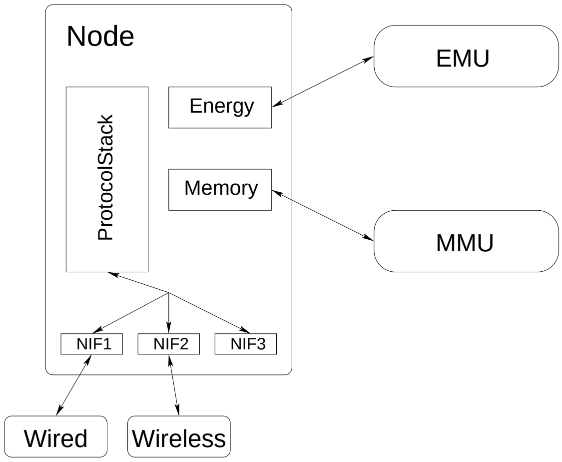

Figure 6.9.: Overview of a node and its attached components

```
receive(event);
if (event->getTime() > local_time){
    usage = EMU->discharge(event->getTime());

    if (usage > maxEnergy){
        state = false;    // Node is dead
    }
}
local_time = event->getTime();
```

Listing 6.3: Decreasing the energy level of a node

vary. But as stated in [LiJo03], energy management also may play an important role in wired networks, e.g. to estimate the power dissipation. Therefore, if some intends to model that behavior, he is provided the possibility to do that in the Cubus architecture.

The second case concerns EMU's of wireless networks. They may incorporate any battery model. We present some in Section 6.8.4. EMU's in this case are very important for managing a node's energy budget. This, however, can be a complex task.

As we have seen in Section 6.8.1, the loss of energy depends on several factors. First of all, the node may be in *idle state*. In this case, the node looses only a small amount of energy. This has to be taken into account. Secondly, all attached hardware devices and network interfaces consume energy. All their individual consumptions has to be summed up and the total has to be subtracted from the node's total amount of available energy. Thirdly, each transmission or reception of packets requires energy and therefore reduces the amount of available energy on a node. Fourthly, operations carried out on the node in general consume energy.

During our examination of energy models in available simulators in Section 6.8.2, we have seen that their approaches are somehow insufficient for our needs. They are not as accurate as we would like to have them. Therefore, Cubus simulates this decrease in energy based on the following principle.

Each time a new event is received by a node or its attached components, i.e. an update in SimTime occurs, the node calculates how much energy it has lost during that period of time. The basic principle is outlined in Listing 6.3.

Since we strike for flexibility by providing the possibility of replacing EMU's during runtime, EMU's must have a common interface. Each EMU incorporates a specific battery model that is used after the EMU is attached to a node. The common interface is defined by providing a `discharge()` method, that is given the current time-stamp of the event being processed.

The EMU then calculates the loss of energy for the interval $[0, timestamp]$ by applying the incorporated battery model. If the current discharge is greater than the batteries capacity, which has been defined at startup as `maxEnergy`, the node is out of energy and therefore is supposed to be dead. A dead node is set to be inactive and thereby does not receive any packets any more, i.e. they are dropped directly at the network interface level.

In addition to the discharge imposed by a batteries non-linear behavior, a node looses energy when transmitting and receiving packets at the network interface level. Therefore, this loss is calculated at transmission time and reception time, respectively. To simulate the loss during transmission, the packet's size and transmission time has to be taken into account. Each EMU therefore provides another discharge method, which expects the packet size for further calculations. We apply Equation 6.2 for these calculations.

This energy management provides a adequate means of modeling a node's energy budget. As we have seen in this subsection, each EMU includes a battery model. In the following subsection we describe the Cubus Battery Model in greater detail.

6.8.4. Cubus Battery Model

As outlined in Section 6.8.1, battery discharges are complex models. Therefore, a good choice with high accuracy would be an analytical battery model as presented in [RaVr01]. But there are several drawbacks with this approach. First of all, the model presented there is highly computational intensive which will cause a dramatic decrease of simulation speed.

Secondly, it requires a *load profile* to model a battery. But such a profile is individual to every situation and battery. Therefore, an accurate simulation requires at least two simulation runs. The first run to get the load profile and the second one to use the battery model.

Due to this problems, a model as introduced by the authors of [HaTi03] is preferred. Their model can be used to simulate a lithium-ion battery. Similar models may be developed to simulate other kinds of batteries. An adapted version of the battery model provided in [HaTi03] is described in Listing 6.4.

The function A is used to calculate the non-linear effect on a battery. The parameters used in this Listing are L as battery lifetime, t_k duration of period k and so on. It is taken from [RaVr01], where an in-depth analysis of battery behavior is modeled using partial differential equation.

The Cubus battery model is comprised by this approach which simulates the non-linear behavior of a lithium-ion battery and an energy consumption mechanism similar to GT-NetS as introduced in Section 6.8.2 in Equation 6.2. An adapted version of the GTNetS approach is pointed out in Listing 6.5.

```
double discharge(Time_t t){
      if (node->getEnergy() == L_{p-1}) { // no change
          L_p = node->getEnergy();
          if (t != 0){
                    S_p = t - 1/f;
          } else {
                    S_p = 0;
          }
          p++;
      }
     S_p = t;
      if (p == 1) {
          return 2 * L_0 * A(t, t, 0, β);
      } else {
          return Σ_{k=1}^{p} 2 * L_{k-1} * A(t, S_k, S_{k-1}, β);
      }
}
```

Listing 6.4: Non-linear discharge of a battery

```
double discharge(Packet *p, Time_t t){
     E_current = node->getEnergy();
     duration = p->Size() / bandwidth;
     Δtime = t_current - t_lastUpdate; // time since last battery update

     // energy loss since last battery update
     // non-linear effect
     loss_nl = discharge(t);

     // loss due to CPU power
     loss_cpu = Δtime * node->getCPUPower();

     // loss during transmit or receive
     loss_action = duration * node->getTXPower();

     E_new = E_current - ( loss_action + loss_cpu + loss_nl );

     return E_new;
}
```

Listing 6.5: Battery discharge during packet transmission or reception

6.9. Memory Management

In addition to limited energy resources, mobile and wireless nodes typically also have limited memory. Mobile and wireless nodes have only a small amount of memory compared to standard computing units, e.g. PCs. One reason is their limited overall size and energy. Memory consumes energy and additionally, the small devices do not have enough space left to contain larger amounts of memory. For example, Smart-Its [TecO04], typical sensornet nodes, have a memory of 13 kbytes. Considering this small amount of memory, the user has to be aware how much memory his program requires.

Therefore, another feature important in an accurate simulation of mobile networks is a decent memory management of mobile nodes. We describe the memory management provided with this simulator in this section.

An accurate simulation should provide the possibility to estimate the memory that is used on each node by the components on this node. Unfortunately, most simulators available for mobile networks do not provide sufficient memory management on nodes. A memory simulation may range from very fine-grained, e.g. almost a complete emulation of the underlying hardware architecture, to coarse-grained simulation, where only the most memory consuming objects are considered. A fine-grained simulation is theoretically the best, but probably would have dramatic impact on the simulation speed. The more calculations have to be done and the more parameters have to be considered, the slower the simulation gets.

Therefore, we suppose that it is best to let the user decide what level of granularity he wants to use. This assumption leads to a plugin solution as in the case of energy models.

A detailed simulation of a node's memory management can be very complex to realize, as pointed out in [ClCS02]. The authors show how to implement a simulator for the whole SPARC V9 architecture. While this architecture is quite complex compared to a mobile node, it still provides some clues how to do that for smaller architectures. Additionally, the underlying memory architecture has a certain influence on the simulated execution of operations on the node, as shown in [ChGr00].

6.9.1. Memory Management in Real Hardware

Before describing the memory management of Cubus, we closely examine how memory is managed in general. The following description is kept as simple as possible but should be sufficient for most cases.

The memory, an operating system provides for an application, is mainly split into two areas (simplified) [AhSU88], [Tane95]: the *runtime stack* which stores the activation records and the *runtime heap* that is used for dynamic objects.

The memory may be used in two different ways, as depicted in Figure 6.10. Situation (a) is primarily used in embedded systems. In such systems memory is very limited, therefore no memory is wasted when heap and stack grow towards each other. But care has to be taken that stack and heap do not overwrite each other.

Situation (b) is frequently found in the rest of the computer world. The stack has a predefined default size. That part of the memory is allocated for the stack solely, while the heap can keep on growing as long as it gets new space from the operating system.

The first approach should be used in a simulator for mobile networks, since it models the internal memory structure of mobile and wireless nodes best.

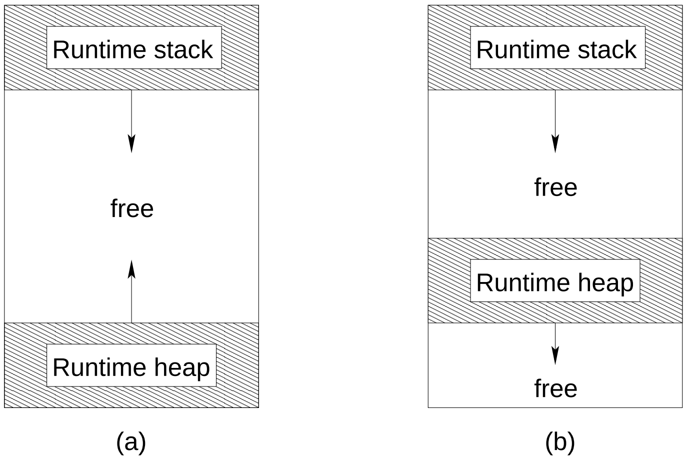

Figure 6.10.: Basic memory management models

Various approaches exist, that model memory systems. The authors of [ChGr00] provide a simple memory model for performance prediction. Their primary focus is on runtime behavior of a large applications. But some of their concepts can be used for simple memory simulation. [PiSM95] shows some techniques on how the memory usage can be measured and determined. They briefly describe techniques how to determine memory usage at runtime. In a survey on memory simulation techniques [UhMu97] possibilities to determine the efficiency of memory architectures are shown.

6.9.2. Memory Management and Models in Existing Simulators

Neither NS-2, GTNetS nor GloMoSim provide the possibility of memory simulation. Although this is an important feature for realistic simulations. Therefore, it is a great improvement if a simulator provides memory management of network nodes. Even a very simple solution results in more accurate results. It allows the estimation of memory usage on nodes. We thereby can determine if applications or algorithms will run on real hardware.

6.9.3. Cubus Memory Management

Cubus uses *Memory Management Units* (or *MMUs*) to simulate a node's memory. Each node owns exactly one MMU, as is the case with EMUs. It can be replaced by another one during runtime. A node does not care about the MMU it has attached. It simply knows the two values of `memory` and `maxMemory`. A MMU reads and changes the first value via the appropriate methods `getMemory()` and `setMemory()`. This provides the possibility that an arbitrary memory management can be used. This situation is depicted in Figure 6.9. It works the same way as for the energy model.

As soon as any changes are made on a node that may consume memory, the associated MMU has to be informed by invoking the appropriate method.

We have previously seen that two basic memory objects exist, the runtime heap and the runtime stack. An accurate memory simulation requires modeling both. We choose situation (*b*) from Figure 6.10 as a reference model for our memory simulation. It is a common approach among real-world mobile and wireless network nodes and therefore fits our needs best.

```
bool decreaseFreeMemory(unsigned int m){
    unsigned int mem = node->getMemory();
    mem = mem - m;
    // check if enough memory is available
    if (mem > 0){
        node->setMemory(mem);
        return true;
    } else {
        return false;
    }
}
```

Listing 6.6: Decreasing free memory

First, we describe the utilization of the runtime heap of nodes. Secondly, we introduce a runtime stack to enhance our memory simulation. We classify these approaches as *memory models*. In addition to the distinction between a heap model and a stack model, we distinguish a coarse-grained heap model and fine-grained one. We introduce them in the following subsection.

6.9.4. Cubus Memory Models

A memory model describes the applied memory. The level of granularity ranges from very detailed simulation, e.g. emulation of the node's underlying operating system or coarse-grained simulation which neglects details in memory allocation and deallocation. The level depends on the MMU that is plugged to the node and on the usage of the provided methods, e.g. when developing a new algorithm.

Although an accurate model is supposed to be very realistic, it probably is quite slow due to extensive checks and updates to the MMU. Therefore, we first discuss a basic, coarse-grained model, before we go into greater detail. The first two models neglect the usage of runtime stacks. After discussing the heap management, we examine the utilization of a stack.

A Coarse-Grained Approach

A coarse-grain approach sets the stack management aside and only considers the heap. This simplifies the management of free memory tremendously. As we will see, heap objects are easier to handle than stacks.

This approach neglects details of the simulated objects. Only complete objects are taken into account. Their internal behavior and memory usage is not examined. This approach allows to use the `sizeof` operator C++ provides.

The MMU is only responsible for decreasing and increasing the node's available memory. Every time a hardware device, a network interface or an algorithm is attached to a node, the corresponding method of the MMU is called, `decreaseFreeMemory()`. Its behavior is outlined in Listing 6.6. It checks if enough memory is left on the node. This model uses the `sizeof` operator of C++ to determine the size of the object that is going to be allocated.

```
void increaseFreeMemory(unsigned int m){
    unsigned int mem = node->getMemory();
    mem = mem + m;
    // not more than maxMemory is allowed
    if (mem > node->getMaxMemory()) {
        node->setMemory(node->getMaxMemory());
    } else {
        node->setMemory(mem);
    }
}
```

Listing 6.7: Increasing free memory

The corresponding increase method is shown in Listing 6.7. It is called whenever a device or algorithm is removed from the node.

This simple approach provides the possibility to check the memory every time something new is added. If there is not enough memory left on the node, `decreaseFreeMemory` returns `false` which prevents adding the device or algorithm.

But there are some major disadvantages with this approach. First, a main part of a real system is not considered, the runtime stack. The stack requires a certain amount of memory in a real system, e.g. to manage program execution. In this coarse-grain approach only the heap is considered which allows more objects to be created in the simulation of a node than it would be possible in the real system due to the lack of the runtime stack in the simulation.

Secondly, the object's size that is to be allocated on the heap is determined on the simulator's architecture. This may falsify the result. For example, if the size of an Integer is determined, the simulator most likely returns a size of four or eight bytes, since the simulator most likely runs on a 32*bit* or 64*bit* architecture. But a mobile and wireless node is probably based on a "smaller" architecture like a 8*bit* machine. There an integer may only be one byte, depending on the compiler used and the underlying hardware architecture.

Thirdly, memory is statically allocated when an object is attached to the node. A variation in memory usage during runtime is not simulated, e.g. when objects are allocated within an algorithm.

But nevertheless, this approach can be used to get a quick overview on memory usage of a node. This technique provides an easy to implement and fairly accurate way of simulating the memory usage on a node.

A Fine-Grained Approach

The approach described in the previous paragraph has some constraints. To get a more detailed view on memory management, a finer-grained examination of the internals of algorithms and devices has to be used. Therefore, some new methods are required:

- `cubusMalloc`
- `cubusFree`
- `cubusNew`
- `cubusDelete`

These four methods provide wrappers for the corresponding C functions and C++ methods. They add the feature of checking and managing memory. We examine the internals of algorithms and devices. They are no longer considered as a black box as in the coarse-grain approach. Memory usage varies during the runtime of an algorithm, e.g. new packets are generated, the packet's size shrinks and grows.

To provide an inside look into the memory variability of algorithms and devices a special field, named `memoryUsage`, is used within them. It contains the current value of memory needed by the algorithm or device. The node's MMU has to check this value each time a change occurs within the object. The MMU then decides if an operation is possible or not. For example, a packet can only be created if enough memory is left on the node. In case of an error, the node is blocked and stops working. A restart of the node's simulated operating system is possible, but depends on the MMUs recovery mechanism.

Every time a new heap object is created on the node, e.g. during the runtime of an algorithm, the MMU's `decreaseFreeMemory` is invoked which then handles further processing. Deletion of heap objects works analogous by invoking `increaseFreeMemory`.

The invocation of the two methods is done automatically, when the methods introduced in this paragraph, `cubusMalloc` and so on, are applied. The user does not need to care about this. He uses these methods at appropriate places instead of the standard operations.

Adding a Runtime Stack

To provide a more detailed memory simulation, a runtime stack has to be added to our model. The main components on the stack are *activation records*. Their size has to be estimated according to the simulated hardware architecture. We assume C or C++ as programming language for algorithms.

In Linux with GCC a runtime stack basically looks like figure 6.11 [Maue03].

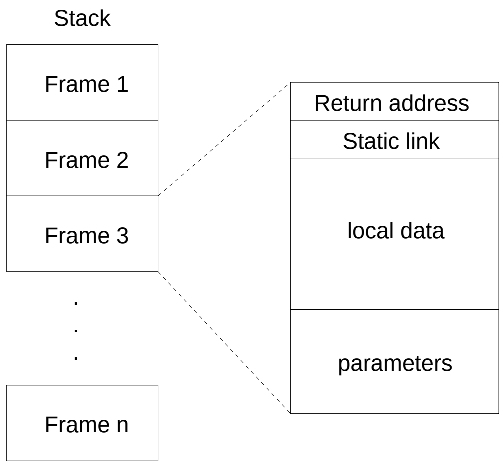

Figure 6.11.: Stack and activation record with GCC and Linux

Some values highly depend on the architecture the simulation runs on. For example, in a 32 bit architecture like $x86$, the return address and the static link each have a size of four bytes. These architecture dependent values have to be set at startup time. They are stored in the MMU, which has a corresponding field that designates the simulated hardware architecture.

But it is more difficult to determine the amount of memory that is required by local data and parameters that were passed to the function. The following description assumes a 32 bit architecture. Minor adaptations on sizes have to be made when a different architecture is used.

First of all, `memoryUsage` has to be decreased by eight bytes as soon as a method is invoked. The eight bytes represent the “static” information in each activation record, the return address and the static link. A method invocation pushes a basic activation record that contains this information on the stack. Both static link and return address require four bytes each. This basic records needs to be extended by investigating the called method.

As soon as the method is entered, its formal parameters have to be checked and their corresponding sizes determined. Their sizes are subtracted from memory, too. Care has to be taken, that stack and heap objects are not mixed.

The size estimation continues for all local variables, that are no heap objects. Memory is increased as soon as that method is left. To realize this principle, the appropriate methods of the associated MMU have to be invoked.

- `addToStack()`
- `removeFromStack()`

The approach described in this paragraph enables the use of a runtime stack in our memory simulation at nodes. But as we have seen, using a stack is much more complex than managing heap objects. The developer of algorithms and devices has to implement the use of stacks himself. He has to extend the methods himself in a way that the `addToStack` and `removeFromStack` are used.

The approach described above is just a simple way on how to simulate the stack behavior. It is quite accurate but lacks some efficiency, since many methods have to be invoked for every object and variable that is created or destroyed. But the granularity can arbitrarily be set.

6.10. Summary

This chapter has described the design of Cubus' simulation components. At first, we discussed algorithms which have a common interface. This interface allows the combination of algorithms and communication between them.

Algorithms are connected to each other in order to build a protocol stack. The connections in this stack resemble a graph. The usage of a graph allows arbitrary connections of algorithms in a protocol stack. Applications form the top layer of each protocol stack. The lowest layer is built by network interfaces.

In our simulator, communication can take place either intra-stack, i.e. communication within the same protocol stack, and inter-stack which in turn allows the communication between protocol stacks of different nodes.

The communication within the same protocol stack is done by using method-based communication. Inter-stack communication is carried out by event-based communication. The transition between method-based communication and event-based communication takes place in the network interfaces which are connected to a medium. A medium in turn models a the physical characteristics of a real-world medium.

Thereafter, we discussed extensions to network nodes since nodes. The extensions consist of hardware devices like positioning devices, memory management units and energy management units. All of them can be used to extend a node's functionality for providing more detailed and realistic network simulations.

We have seen that a simulation of a node's hardware architecture is important. The architecture of a node influences the speed packets can be processed on the node. We discussed the principle problems that arise when simulating the hardware architecture of a node in a sequential simulator, e.g. simulating of parallel processes in the node are hard to realize.

Chapter 5 and Chapter 6 describe the design of Cubus. They form the simulation engine, the node engine and the algorithm engine as they were introduced in Chapter 4.

7. Implementation

Chapters 5 and 6 described the basic design of our simulator's architecture. This chapter provides a look at the prototype implementation of our approach. We show the aspects that were implemented and those that have not been fully implemented yet.

This chapter does not include any source code. For more detailed information about the implementation itself, refer to the code which is well documented.

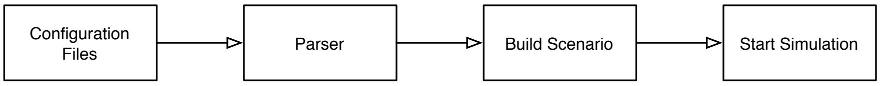

Figure 7.1.: Process flow of a simulation run

Figure 7.1 depicts the principle process flow of a simulation. We first discuss the implementation and parsing of the algorithm configuration files and the scenario file. Thereafter, we show how the scenario is built up. Finally, we examine the simulation run itself.

7.1. Preliminary Notes

Our simulator is implemented in C++ [PrKP02, Oual02]. Due to portability reasons, care was taken to stay as close as possible to the ISO C++ Standard. To simplify and speed up some part of the implementation and the development process, STL templates [Sili04] are used.

Among other things, the choice of the programming language is made in favor of C++ due to performance reasons. But C++ caused some problems, too. The lack of reflection, as described in Section 6.1.1, makes things more difficult. The lack of reflection forces us to find some way of describing the internal structure of an object. Therefore, we introduce configuration files, like the algorithm configuration files.

Another problem was dynamic loading during runtime. C++ lacks the ability to dynamically load objects during runtime. C uses functions like `dlopen()` to load functions dynamically during runtime. Unfortunately, `dlopen()` and similar functions are not able to handle object-oriented concepts [Nort00]. Therefore, we need to write wrappers for these functions in order to support object-orientation.

7.2. Configuration and Scenario Files

As we have seen in Chapters 5 and 6, we distinguish two types of files: configuration files and scenario files. This is depicted in Figure 7.2.

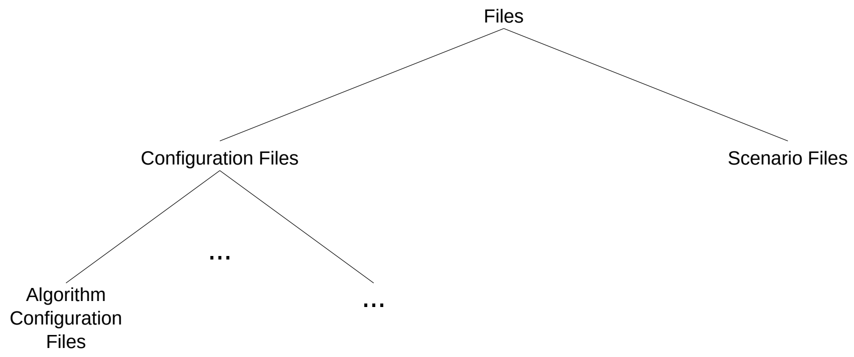

Figure 7.2.: Classification of files in our simulator

7.2.1. Configuration Files

The configuration files are required for describing dynamically loadable objects. Each object that is supposed to be loaded dynamically requires one configuration file, that describes its internal structure. The basic layout of configuration files is described in Appendix A.2.

Our prototype implementation of Cubus only supports algorithms and applications as dynamically loadable objects and thereby, only algorithm and application configuration files are supported.

All algorithm configuration files are stored in a special directory called `algorithm`. Each algorithm has its own configuration file which defines the default values for its parameters and describes all of its custom parameters, as described in Section 5.4 and 6.1.1.

A rudimentary algorithm configuration file is outlined in Listing 7.1. The first four parameters `name`, `type`, `location` and `loglevel` are mandatory. The `property` construct describes a custom parameter. An arbitrary number of these constructs can be used in each configuration file.

7.2.2. Scenario Files

As described in Section 5.5.1, scenario files describe a whole scenario, i.e. all nodes, their protocol stacks and the way the nodes are interconnected. The basic principle of a scenario file is outlined in Listing 7.2, a concrete example is given in Appendix A.3.

```
name = ...;
type = ...;
location = ...;
loglevel = 5;

property{
   name = customParam1;
   min = 5;
   max = 25;
   step = 1;
   editable = true;
};
```

Listing 7.1: Basic algorithm configuration file

```
cinterfaces{
    Interface location inf1;
};

cmedium{
    Wlan location med;
};

cnode{
    name = node1;
    ...
    cprotocolStack{
        Algorithm "../examples/libdummy1.so" a;
        Algorithm "../examples/libdummy2.so" b;
                cregister(a, b);
    };
};

cregister(inf1 , a);

sharedLink {
    med -> node1(inf1);
};
```

Listing 7.2: Basic scenario file

In this prototype implementation the scenario file can be given as a command-line argument and the startup of the simulator. In addition to the description of the simulated network topology and its components, we can use events in the scenario file to start and stop the simulation. This restriction to start and stop events only exists in this prototype implementation. We do not provide the use of other events in the scenario file yet. This requires some minor adaptations in the way the parser processes the file.

We have described configuration files and scenario files in this section. After these files are created, we can hand it over to our simulator. By doing this, the parser comes into account. The parser is described in the following section.

7.3. Parser

The parser is the next step in the process flow. It analyzes the configuration files, sets up the corresponding repositories and builds the scenario by analyzing the scenario file. First, we outline the creation of the parser. Secondly, we describe how the configuration files are processed. Since the build process of scenarios is more complex than the handling of configuration files, we cover it in Section 7.4.

7.3.1. Creating the Parser

Basically, two types of parsers are available, *bottom-up* and *top-down* [GrJa98]. Memory efficiency and the availability of lexer and parser generators suggest the use of a bottom-up approach. Top-down parsers typically use the method of "recursive descent" [AhSU88] for parsing. This requires a larger amount of memory, since most parts of parsing process have to be kept in memory.

Therefore, the lexer generator *flex* and its parser counterpart *yacc* [LeMB95] are used. The according LALR(1) grammar, that describes our configuration files and scenario files, is listed in Appendix A.

By using the grammar the parser has been generated and appropriate methods for handling the parsed files have been added. The following subsection describes how the files are handled.

7.3.2. Handling the Files

The basic process for reading the configuration files and the scenario file is depicted in Figure 7.3. As mentioned in Section 7.2.1, only algorithms can be dynamically loaded in this prototype implementation during runtime and therefore require configuration files. But the basic principle outlined in Chapter 5 can be used to adapt this feature to the other components that should be loadable during runtime.

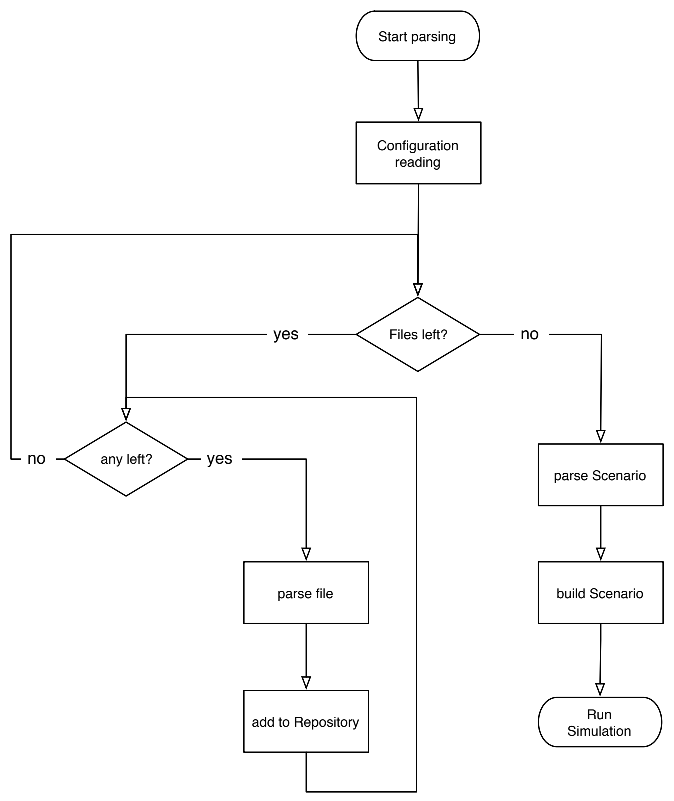

Figure 7.3.: Process of parsing files

Configuration files for algorithms are stored in a special directory, named `algorithm`. As soon as Cubus is started, all files in this directory are automatically parsed and the corresponding values are stored in the *Algorithm repository*, as described in Section 5.4. For each configuration file found in the directory, i.e. for each type of algorithm, a new entry is added to the *algorithm repository*. The entry can be identified by the type of the algorithm. In addition to the type information, the entry also contains information about all parameters used in the associated algorithm.

If Cubus is extended to support dynamic loading of other components, the corresponding configuration files have to be stored in specific directories like `algorithm` in our case.

As soon as all configuration files have been read, the scenario is parsed. The following section describes how a scenario is built from the scenario file in our prototype implementation.

7.4. Building the Scenario

This section describes the built-up process of an scenario. To understand this process, we examine how the simulation components (see Chapter 6) are implemented in this prototype of our simulator. We first describe these components. Then, we examine the built-up process itself.

7.4.1. Algorithms and Protocol Stack

In Chapter 6 we have seen that algorithms require one common interface. This is necessary to allow exchanges and dynamic loading of algorithms during runtime.

The common interface is mainly responsible for two things. The first part is responsible for interconnecting algorithms and thereby constructing a protocol stack. The second one allows two neighboring algorithms to communicate. We describe our implementation approach of interconnection capabilities first.

As we have seen in Section 6.2, the graph oriented approach for connecting algorithms is preferred. To connect two algorithms, a method `register(Algorithm*)` exists. By this means the calling algorithm is connected with the algorithm given as parameter. The connection is bidirectional, i.e. packets can be passed in both directions, from the lower layer algorithm to the upper layer one and vice versa. When two algorithms are connected, the corresponding upper and lower fields of both algorithms are updated. In our implementation these fields contain references to the corresponding algorithms.

For communication purposes corresponding `send` and `receive` methods exist. If we want to send a packet to a lower layer algorithm, two possible types of send methods exist. First, `send(Packet)` hands the packet over to all available lower layer algorithms. Secondly, the method `send(Packet, Algorithm)` gives the packet to the specified lower layer algorithm, i.e. the second method is a *selective send*. When sending a packet appropriate packet headers are attached. The principle outlined in Section 6.2.2 regarding method-based communication is implemented.

The `receive(Packet)` method always calls the correct upper layer algorithm. The correct algorithm is identified by the current packet header.

In this thesis we provide two algorithms, a simple flooding algorithm and an implementation of IEEE 802.11. The flooding algorithm is described in Chapter 8 in detail. However, the implementation of the IEEE 802.11 MAC layer has been ported from GTNetS. It was enhanced to fit our algorithm interface and to make it dynamically loadable during runtime. The implementation provided resembles the IEEE 802.11 Standard [IEEE99a] including the RTS/CTS functionality.

7.4.2. Network Interfaces

As we have already seen in Section 6.4 network interfaces are important for modeling the communication between nodes. In this thesis we have implemented a network interface base class which encapsulates all necessary features like the capability of handling events and of interacting with algorithms via method-based communication.

Since the focus of this thesis is on mobile ad hoc networks, we have implemented a very simple *Wireless Interface* that is derived from the network interface base class. It is only a basic model with rudimentary features. This includes a MAC address and thresholds for receiving and transmitting packets. The thresholds model the behavior of real-world antennas. They are used by propagation models.

7.4.3. Medium

We have implemented a medium base class according to the properties described in Section 6.5. Additionally, we provide two media in this thesis, a wireless medium and a wired medium. Both have been partially ported from GTNetS and enhanced to fit our needs. An appropriate error model is added to both of them.

Our medium base class defines an interface that allows adding arbitrary propagation models. This interface allows switching propagation models "on the fly". The only thing a medium has to know about a propagation model is its *transmission range query*. Its signature is common to all propagation models.

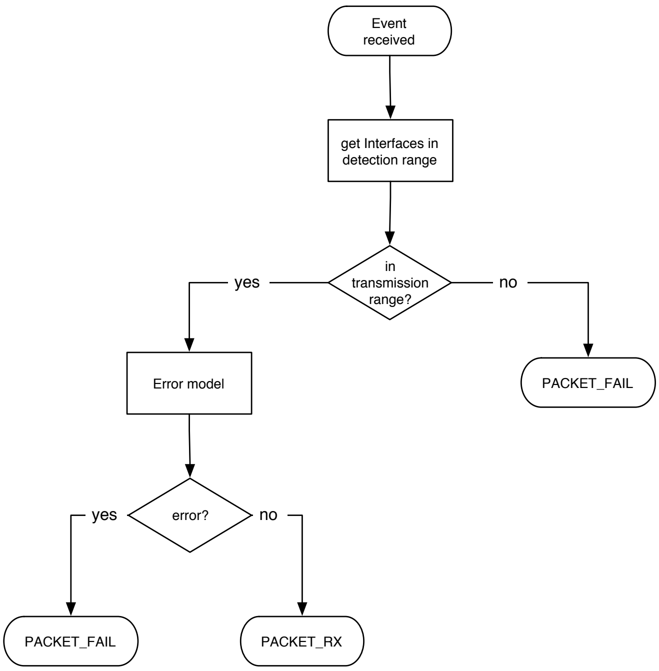

Figure 7.4.: Packet transmission and receiption

Figure 7.4 depicts the process of transmitting a packet on a wireless medium. First, we retrieve all network interfaces in detection range when a packet reaches the medium. The detection range is determined by calculating the *Euclidean distance* between the sender and all network interfaces attached to the same medium. The Euclidean distance is given by Equation 7.1.

$$\text{distance} = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} \tag{7.1}$$

Secondly, a propagation model determines those network interfaces that are within transmission range. The calculation of the transmission range can be very complex, i.e. it may consider environmental effects like reflection at obstacles during the calculation. However, propagation models are not part of this thesis. For evaluation purposes we have implemented a simple propagation model based on free space path loss.

After the network interfaces in transmission range have been determined, some of them are marked defective. This roughly models the error-proneness and unreliability of real-world media like air.

7.4.4. Node

Nodes, one of the base classes of our simulator, contain all necessary information required for an accurate simulation. Most parts we have described in Section 6.6 are implemented.

Thus, nodes have parameters for the available memory (`memory`), the available energy (`energy`), the CPU speed in BogoMips (`cpu`) and the information of the architecture (`architecture`).

In addition, references to all algorithms of its own protocol stack, one EMU and one MMU are included. These are necessary for a proper operation. Appropriate access methods are provided.

The parameter, `state`, shows if the node is active or dead due to some reason. Each time a node's network interface receives an event, i.e. if receiving a packet, this parameter is checked. If it is set to `false`, the node is dead and the packet dropped. This can happen due to a lack of energy.

The hardware simulation, as described in Section 6.6.3, is only partially implemented. We discuss this in Section 7.5 in greater detail.

7.4.5. Energy and Memory Management

We have implemented two different energy models in this thesis. Each model is encapsulated in its own EMU. The first EMU is similar to that provided with the original NS-2 version. Using this model, each transmit and receive operation at a node's network interface reduces the amount of available energy by $30mW$. This is a fast but not very realistic and accurate energy model. Nevertheless, it is sufficient for testing purposes where the behavior of a node's energy source is not very important. We call this EMU with its energy model `SimpleEnergy`.

The second model resembles in some parts the GTNetS model when transmitting and receiving packets. Additionally, we take the non-linear effects of batteries into account. The model of non-linear effects of batteries have been adapted from [HaTi03]. As discussed in Section 6.8, this model is by far the most realistic we have implemented yet. It models the behavior of a real-world energy source for mobile nodes in a fairly accurate way.

Each time a new event is received by a component that is attached to the node, `discharge(Time_t)` is called. This method of the EMU updates the energy level of the node. As discussed in Section 6.8, this models the non-linear behavior of the battery. In addition to this, we invoke `discharge(Time_t, Packet)` each time a packet is transmitted or received at a network interface of the node. This behavior is depicted in Figure 7.5.

This basic principle holds for both the simple model and the more advanced one. Whereas in the former case `discharge(Time_t)` does not have any effect on the energy level and `discharge(Time_t, Packet)` simply decreases the energy level by $30mW$ whenever invoked.

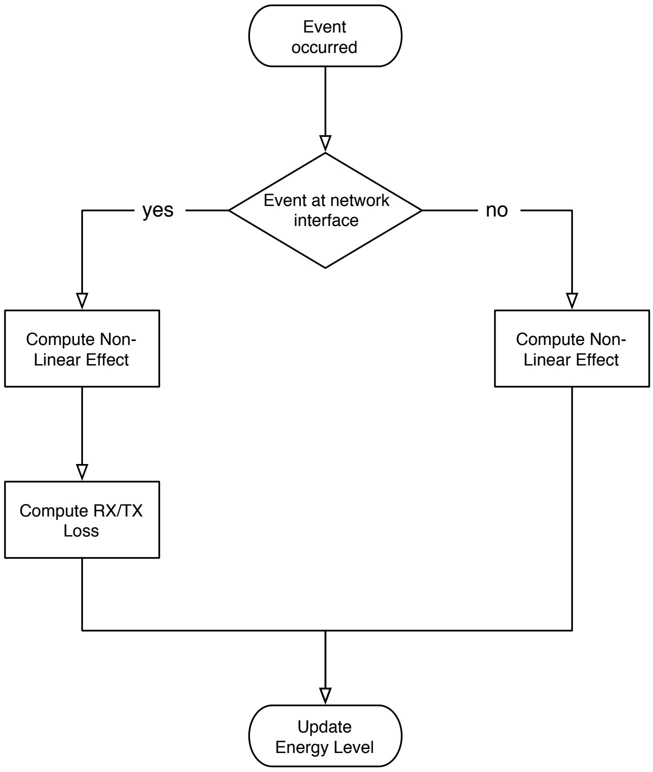

Figure 7.5.: Energy Computation

Only a very basic memory model has been implemented during the course of this thesis. It its roughly equal to the coarse-grain approach introduced in Section 6.9. There have been some problems with the use of `cubusNew`, etc. These methods are a bit pedestrian to use.

7.4.6. Building the Scenario from the Components

The previous subsections described the implementation of the simulation components of our simulator. In this subsection, we discuss the way of building the scenario by using the scenario file and the simulation components.

During the parse process methods are invoked which build the scenario by instantiating its components. In addition, these methods also set the values specified by the user in the scenario file. We need some central components of global and static scope that can be accessed by all other components. These central components are called *Containers*.

As stated previously, we require containers since other components, especially the core engine, need to have access to all components. The core engine manages all components of a simulation scenario and therefore requires access to them.

Therefore, we provide specific containers for each simulation component in our prototype implementation. We use containers for storing algorithm instances, node instances and so on. Whenever the parser detects a simulation component in the scenario file, e.g. a network interface, Cubus instantiates an object and provides it with the values given in the scenario file. Then, the instantiated object is stored in the appropriate container. This process is depicted in Figure 7.6.

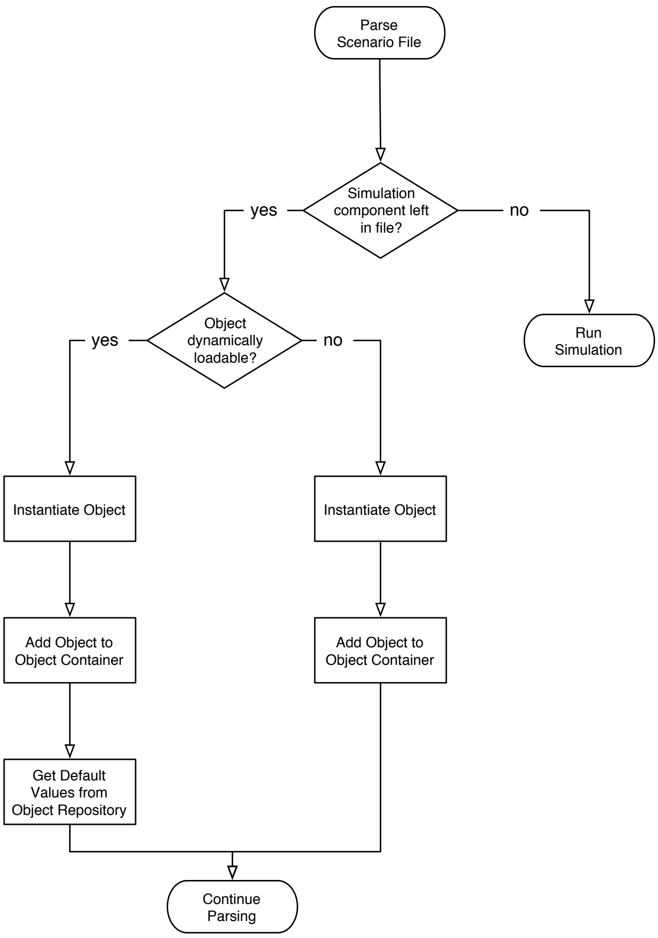

Figure 7.6.: Object instantiation during parse process

During the creation of nodes, their protocol stacks are created, too. Whenever an algorithm is instantiated, Cubus uses the algorithm repository to look up the corresponding default values for this algorithm and assigns these values to this instance. Thereafter, the components are connected with each other, as defined in the scenario file. After the scenario is set up, one ore more start and stop events are generated and scheduled in the simulation queue, according to the scenario file. Our prototype implementation supports only these two types of events.

7.5. Simulation Run

The description how a complete simulation run is implemented and is carried out is beyond the scope of this thesis. The interactions within a run can be far to complex, so that they cannot be described in this document. Therefore, we describe some aspects regarding event processing in this section.

As soon as the parse process is finished, the scenario built-up and the start event scheduled, the simulator can be started by invoking its `run` method.

In the course of the simulation, many events are scheduled and processed. As we have seen in Section 5.3.2, Cubus employs a publish / subscribe system. This system is fully implemented in our prototype implementation. It works as follows.

The simulator manages a container of all available event types. Each type has an associated list of subscribers within the container. Whenever an event is taken from the simulation queue, it is checked whether an subscriber exists for this event or not. If so, the subscriber is informed. Then, the simulator continues to process the event by updating the SimTime and invoking the associated event handler.

The event execution principle, outlined Section 5.3.3 is fully implemented in the prototype, we provide with this thesis. Finally, the same holds for the event queues. Although, in most cases the simulation queue is used.

7.6. Summary

In this chapter, we presented a prototype implementation of our simulator. We first focussed on configuration and scenario files. We support dynamic loading of algorithms in this thesis. Therefore, we use algorithm configuration files to handle custom parameters of these objects.

Whenever a algorithm is instantiated during the parsing process, the default values of the corresponding algorithm type are looked up in the algorithm repository and thereafter set to these default values. However, the parameter repository is not implemented yet. Only the algorithm repository has been implemented in this thesis.

All simulation components that were designed in Chapter 6 have been implemented in this prototype. However, some problem exist with the implementation of the memory management units.

Furthermore, we provide the implementation of the design concepts of our simulator's core engine in this diploma thesis. This prototype implements the publish / subscribe mechanism and the event execution methods.

However, the implementation of this prototype shows that our design ideas work. We will evaluate our simulator in the following chapter by comparing it to other simulators. For evaluation, we will use a simple flooding scenario.

8. Evaluation

In Chapters 5 and 6 we have described the design of our simulator in detail. In Chapter 7, we have examined its implementation aspects. This chapter contains measurements of Cubus' performance. If reasonable, these results are compared to other simulators, mainly to GTNetS and NS-2.

8.1. Goals

The goals of this evaluation are the determination and assessment of our simulator's performance and behavior. Miscellaneous performance aspects are analyzed. One part of these measurements is its runtime behavior, i.e. the time (WCT) needed by Cubus to process a specific scenario.

During one run measurements of memory consumption, the amount of packets and events are done. The last two values allow predictions on the simulators efficiency, i.e. the fewer events and packets are required, the faster the simulation can run. We describe the metrics of our simulation in greater in the following section.

To get a certain expressiveness of the results, they need to be compared with other simulators. However, this is only reasonable in specific cases. Whether a comparison is reasonable or not is discussed in the appropriate sections. The reference simulator in this chapter is GTNetS, since it was designed especially for wireless networks. Therefore, it does not have all that ballast of wired networks like NS-2 does.

Nevertheless, comparing simulation results of different simulators is difficult. The authors of [CaSS02] tested three different simulators, namely the OPNET Modeler [OPNE04], NS-2 and GloMoSim. In all three cases they used the same scenario with simple flooding. They found out that the results did not only differ quantitatively but also qualitatively in some cases. In the qualitative case, the simulators sometimes differed in their behavior tremendously when simulating the same scenario. The same scenario with the same parameters sometimes behaved entirely different. The authors of [CaSS02] were unable to compare these results, although all three simulators were supposed to do the same thing.

This limits the expressiveness of our results. But it is emphasized in our tests that all simulators under test have the same initial situation if possible.

8.2. Evaluation Metrics

As outlined in the previous section, we distinguish several evaluation metrics. They can be grouped into two categories. The first one, is used to evaluate the performance of our simulator compared to other simulators. The second category examines the behavior of our flooding algorithm compared to the same algorithm on different simulators.

Runtime and memory evaluation belong to the first category, as well as the measurement of packets and events. Latency is part of category two.

We first evaluate the runtime performance of our simulator. We, thereby, compare it to NS-2 and GTNetS. The first measurement concerns the setup time of our simulator. This value denotes the time required to parse the scenario and build it up. It ends just before the simulation itself is started, e.g. by invoking the simulator's run method.

The second metric is the runtime of our simulator, i.e. the time required by the simulator to carry out the simulation. This is necessary to evaluate the performance of our simulator. The faster the simulator is, the more the usability increases.

Another important metric is the memory usage of our simulator. We distinguish two different points of time, when measuring the memory usage. The first one is just after the start of the simulator before anything is done. The processing of the scenario file and thereby the construction of the scenario has not started yet. This measured value denotes the minimum memory required by the simulator. The second measurement takes place right after setting up the scenario, i.e. the complete scenario has been loaded into memory. This shows the size of the initialized scenario. This memory usage should be as low as possible. The larger the scenario grows, i.e. the more nodes are simulated and the more complex the scenario gets, the more memory usage increases. But this increase should be as slow as possible because when memory usage increases slowly, the larger the scenarios can be.

The number of packets and events used in a simulation gives a hint on performance. If many packets and events are used, the memory usage increases and the simulation slows down, since more events and packets have to be allocated and processed. Therefore, we measure these two values with increasing size of our scenario.

Finally, we measure a value to evaluate the flooding algorithm we have implemented in this thesis. This value denotes the latency of our network, i.e. the average simulation time that is needed until a node receives the packet that is flooded for the first time. We need this measurement to validate our simulator. It is not sufficient to measure only the performance aspects of our simulator compared to other simulators. We also need to evaluate whether our simulator works correctly or not. We do this by comparing the latency in our network as imposed by the flooding algorithm we use.

Now, that we have defined and explained the metrics of our evaluation, we introduce the experimental setup used for evaluation purposes.

8.3. Experimental Setup

This section describes the experimental setup used by our simulations. The setup ranges from the hardware used to scenario descriptions. First, we have a look on the hardware, the following subsections describe the scenarios.

All simulations are carried out on an $x86$ architecture. The PC used is a *dual processor* machine equipped with two Intel Pentium IV 1GHz processors and one gigabyte RAM.

8.3.1. Scenario

The following scenario examines Cubus' basic performance characteristics. The simulation covers an area of 1000×1000 meters. Whereas, to simplify things, it is assumed that no obstacles are in this area and that all nodes that are within transmission range can communicate with each other without transmission errors. Obstacles are modeled by the *environmental model* which is part of another thesis.

```
upon receive(m) by Node_i:
    if (alreadySeen(m)){
        delete m;
    } else {
        deliver(m);
        send(m) to all neighbors;
    }

upon broadcast(m) by Node_i:
    send(m) to all neighbors;
    deliver(m);
```

Listing 8.1: Simple flooding

The nodes are supposed to stay at fixed positions. They are all attached to the same medium, a simple Wireless LAN. Each node's transmission range is set to 250 meters. For each simulation run, the number of nodes is increased. The first run is done with 10 nodes, the final one with 5000. We start by increasing the nodes by 10 between each run.

Each node's protocol stack consists of the same components, a *flooding application*, a *simple flooding* algorithm (see Listing 8.1, adapted from [CaSS02]), and a MAC-layer based on IEEE 802.11. The communication stack layout is depicted in Figure 8.1.

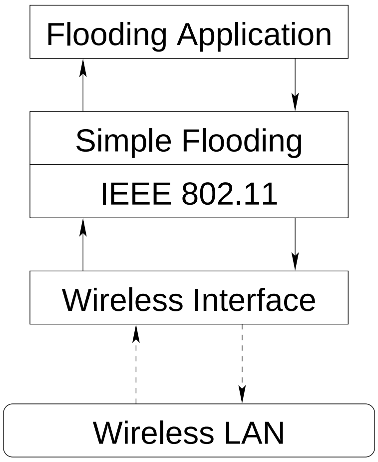

Figure 8.1.: Protocol stack for scenario

The course of this simulation is as follows: The x nodes of each simulation run are randomly distributed (uniform distribution) in the simulation area which covers 1000×1000 meters. *One* randomly chosen node starts sending a packet as soon as the simulation is started. The start time is given by the start event in the scenario file. The simulation runs until no more packets are transmitted, i.e. all reachable nodes have received this packet.

8.4. Simulator Runtime

This section examines the runtime behavior of our simulator. First, we examine the setup time of our simulator compared to NS-2 and GTNetS. Then, we evaluate the runtime performance of our simulator from the time the simulation is started until it is finished, i.e. the last reachable node is flooded.

During this runtime analysis, we compare Cubus to GTNetS and NS-2, since GTNetS and Cubus follow similar approaches whereas NS-2 is the quasi-standard in network simulation. The comparison between the different simulators is done using a relatively small amount of nodes, i.e. 100 nodes. Afterwards, we examine the behavior of Cubus with larger scenarios in greater detail.

8.4.1. Setup Time

Figure 8.2 depicts the setup time of Cubus, NS-2 and GTNetS. The latter one outperforms our simulator and NS-2 clearly.

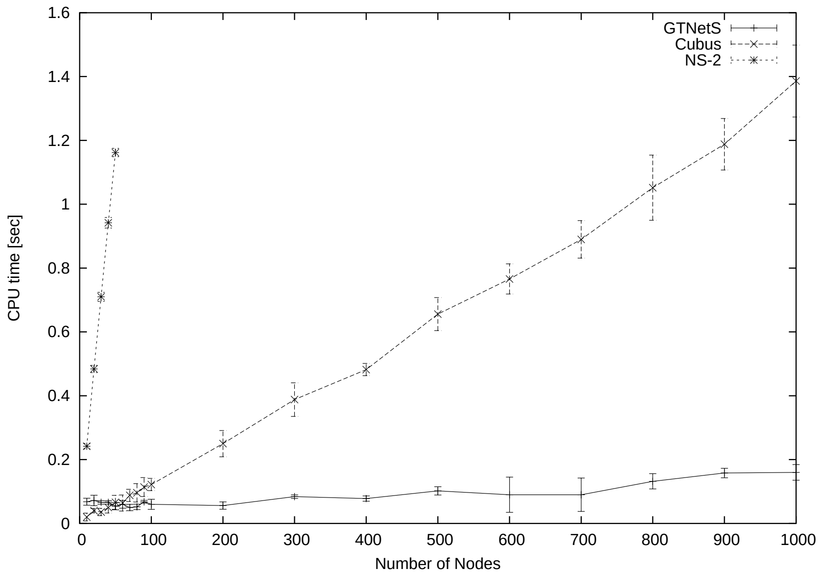

Figure 8.2.: Setup time compared to GTNetS

However, this depends on the way a simulation is created. While GTNetS uses a scenario that is described in the simulation's `main` method, Cubus handles scenarios by configuration files. The use of files requires the use of an appropriate parser. A good parser should not need more than linear time for reading the file [AhSU88].

That is exactly what one can see in Cubus' graph. Doubling the scenario size approximately doubles the setup time, as predicted by theory. For GTNetS, setup time remains almost stays constant when varying the scenario size.

Interestingly, the setup performance of NS-2 is poor compared to Cubus and GTNetS. The main reasons for this behavior of NS-2 probably are the split programming model of NS-2, the use of OTcl and the calculations NS-2 does when setting up a scenario. Both the split programming model and OTcl slow down the setup process. Additionaly, NS-2 does some additional calculations during the setup, i.e. it precomputes certain network characteristics during the setup. Among them are the number of hops to reach nodes and many others. This slows down the setup process. Therefore, NS-2 shows quadratic behavior in setup time.

Both GTNetS and Cubus only build the scenario and do not precompute anything during the setup process.

8.4.2. Runtime Comparison

After examining the time required for setup, we measure the runtime behavior of Cubus, GTNetS and NS-2. The scenario was executed in the way described in Section 8.3.1. The number of nodes was gradually increased from ten to one hundred nodes in steps of ten nodes. We only examine scenarios up to 100 nodes, since the results drift apart to quickly with scenarios larger than 100 nodes. This especially holds for GTNetS.

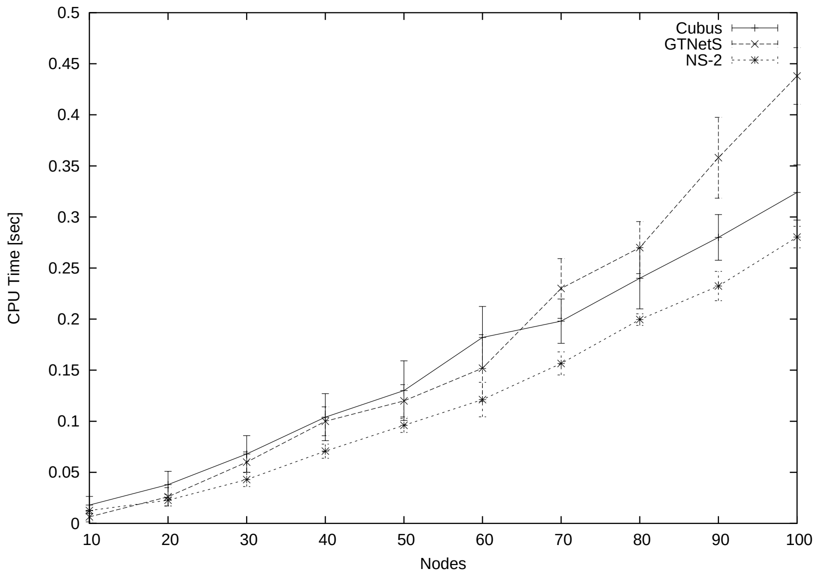

Figure 8.3.: Runtime compared to GTNetS and NS-2

Figure 8.3 plots the number of nodes against the runtime in CPU time. The CPU time specifies the time the program is actually processed by the CPU.

As can be seen, all three simulators require a longer period of time to calculate the simulation with an increasing amount of nodes. The exact numeric values can be found in Appendix B.1 for Cubus and in B.2 for GTNetS.

The increasing duration of the simulation is because of the greater amount of messages that are required to flood to all nodes. Thereby, more collisions and errors occur which increases the simulation's runtime.

The runtime behavior of GTNetS is better than that of Cubus as long as the number of nodes is approximately below 65. Then, Cubus turns out to be faster. The difference grows steadily. GTNetS seems to increase in quadratic order while Cubus stays linear. The runtime behavior of NS-2 is similar to that of Cubus in this simple scenario for scenarios of about 100 nodes. Figure 8.4 depicts the runtime behavior of scenarios with up to 300 nodes. As we can see, NS-2 gets slower at scenario sizes of about 140 nodes. Figure 8.5 shows larger scenarios with up to 1000 nodes. Figure 8.4 is a cutout of 8.5.

However, one has to consider the following things when examining this result. First, as stated in [CaSS02], it is difficult or even impossible in some cases to compare the results of different simulators, since they may vary significantly. Although we take care that most problems have been eliminated before measuring some remain. Among these are the following:

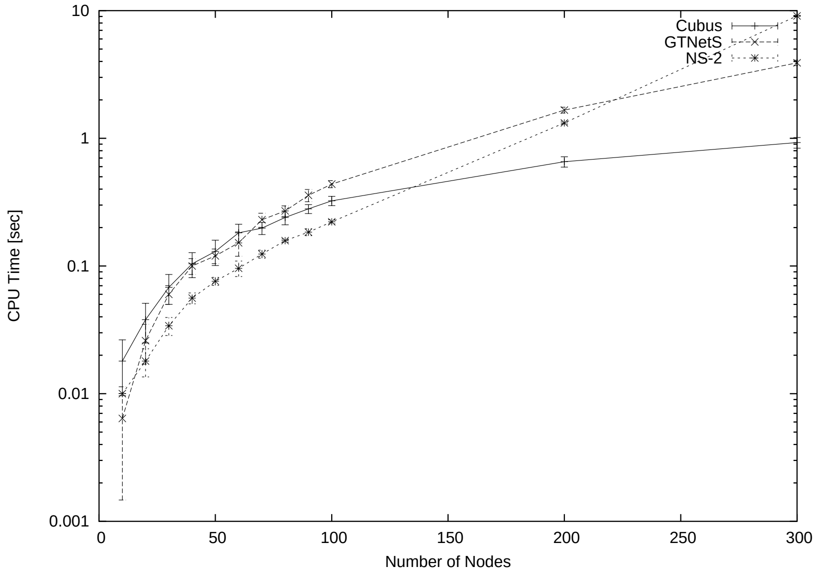

Figure 8.4.: Runtime compared to GTNetS and NS-2 with scenarios of less than 300 nodes

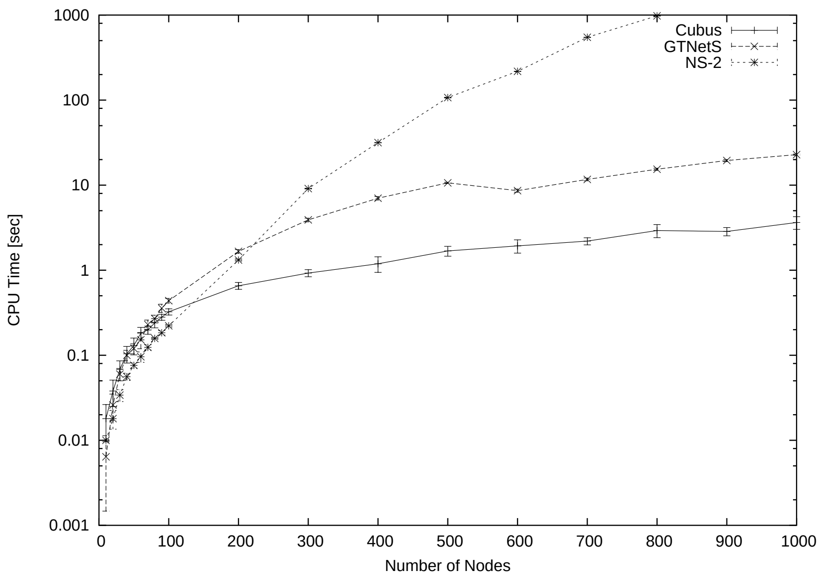

Figure 8.5.: Runtime compared to GTNetS and NS-2 with scenarios up 1000 nodes

First of all, our simulator is not completed yet. It still lacks several important features (see Chapter 10 for more details). Both GTNetS and NS-2 have been available for a longer time now. Therefore, more parts have been fully implemented. Some models are more detailed than the corresponding ones in Cubus. For example, GTNetS wireless interfaces are more realistic than those of Cubus.

Additionally, our simulator relies on a very simple propagation model, Free Space Propagation. The integration of more accurate propagation models are part of a future thesis.

Therefore, with continuing integration of more precise models the graphs will converge a bit. But Cubus should stay in linear order, since some approaches are more efficient. For example, the mixed method- and event-based communication is faster than the pure event-based communication of GTNetS.

Cubus Runtime

We have seen how Cubus behaves compared to GTNetS and NS-2. Figure 8.6 depicts the runtime performance of Cubus for larger scenarios.

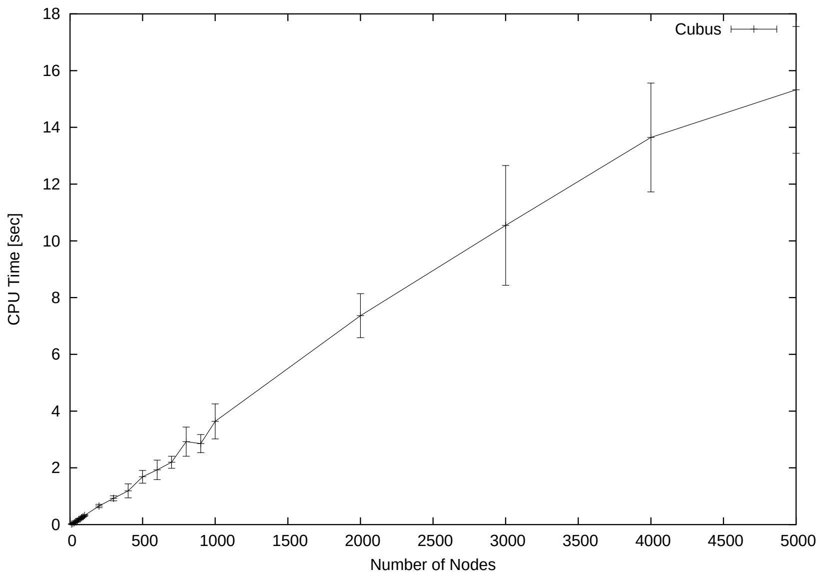

Figure 8.6.: Runtime in CPU Time

It is clearly visible that the runtime behavior is in linear order. The error bars depend not only on simulation specific variations during runtime but also on the scenarios themselves. Since nodes are placed randomly in the simulation area, it sometimes happens that only few nodes are within transmission range of each other. This decreases the runtime. Only a small subset of nodes is flooded. To minimize this effect, for each number of nodes five different runs were simulated after which all nodes were flooded. The graph shows the average of these runs with the corresponding standard deviation.

The standard deviation increases with a larger number of nodes. Because of the nodes being randomly placed, it may happen that all nodes of a scenario are closely together. This reduces the overall runtime, since fewer communication is necessary to flood the packet to all nodes. The scattering of the nodes may vary from run to run and thereby the runtime of our simulator varies.

8.5. Memory Usage

The next set of measurements is concerned with the memory consumption of the respective simulators. Two different situations are analyzed. The first one is the simulator's memory consumption at startup time, i.e. when the simulation has not started yet. The second one is the point of time when the whole scenario has been built and the simulation is ready to start. In this simple scenario, the amount of additional memory that is required during the run is supposed to be small, since only a few packets and events are created. Packets and Events are much smaller than nodes and the other components and therefore do not carry weight.

8.5.1. Memory before Setup

Figure 8.7 depicts memory usage of Cubus and GTNetS at startup time. As assumed, our simulator's memory usage is below that of GTNetS. This is because many simulators, like GTNetS, load all components into memory during startup, even if not all of these components are required for the scenario to be simulated.

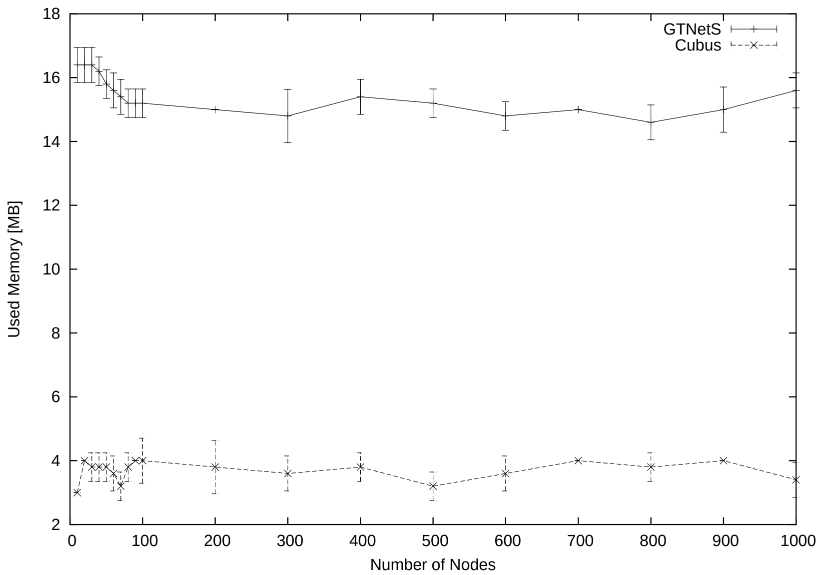

Figure 8.7.: Memory at startup compared to GTNetS

As we have seen in Chapters 5 and 6, Cubus follows a different approach. It only loads those components into memory that are really required for the specific scenario.

Since no information about the scenario is available during startup time, only those parts are loaded into memory that are needed by the simulator for operation. This explains the difference in the base loads of the two simulators that were examined. Our simulator's base load is about four megabytes, whereas that of GTNetS is almost 3.6 times higher.

The base load should not change when varying the number of nodes, since the simulator does not know anything about them at startup time. This explains why the memory consumption of both simulators is almost constant.

8.5.2. Memory after Setup

The situation looks different after the scenario has been built-up and is ready to run. Figure 8.8 shows the situation immediately after the scenario was setup. Theoretically, Cubus' memory usage should be lower than that of GTNetS. Only necessary parts are located in the memory.

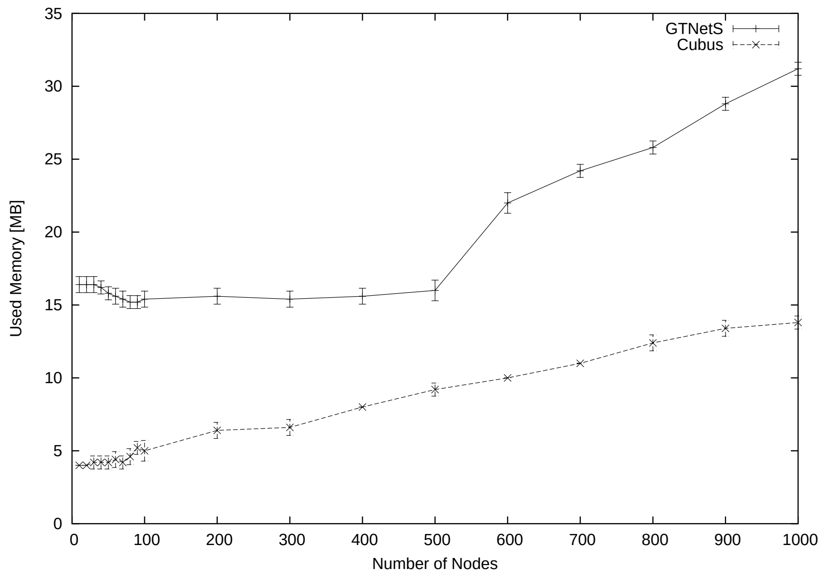

Figure 8.8.: Memory after setup compared to GTNetS

Looking at the figure makes this clear. Cubus only loads some components, like a Wlan medium, n nodes, each populated with a Flooding Application, a simple flooding algorithm, IEEE 802.11, and a network interface for interconnecting IEEE 802.11 and the medium. Therefore, the memory usage is kept small.

Our simulator's memory usage grows linearly, i.e. with increasing scenario size, the memory needs increases, too. In contrast, GTNetS memory consumption is constant at the beginning and it starts growing in linear order at about 500 nodes. This results from the fact, that at the beginning of the simulation there is sufficient amount of memory allocated by GTNetS, so that it can store small scenarios in that part of the main memory.

8.6. Packets and Events

In the case of packets and events, a comparison between Cubus and other simulators is not reasonable. Most discrete event simulators use events as their primary way of communication. Our simulator uses mixed method- and event-based communication which reduces the amount of events required. As mentioned in Section 6.2, communication in each node's protocol stack is based on method invocations.

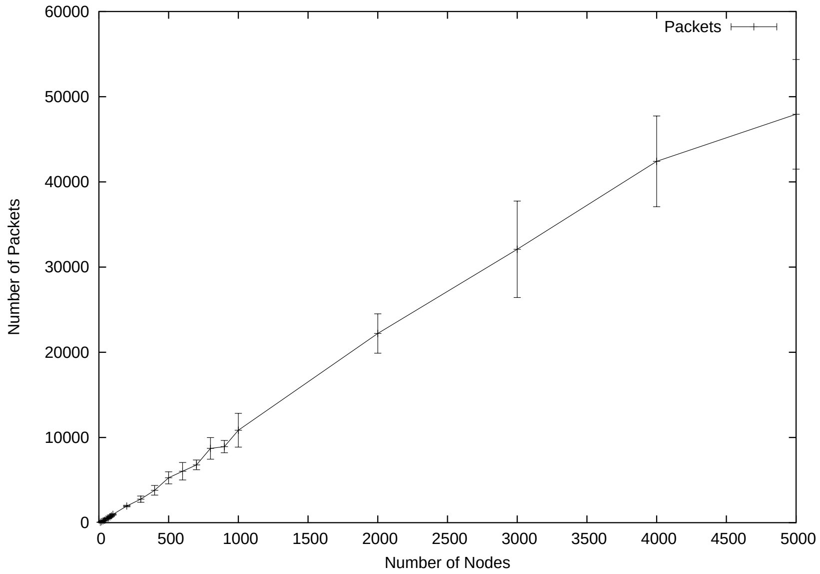

Figure 8.9.: Number of packets with increasing number of nodes

Additionally, the amount of allocated packets varies, too. For example, it depends on the number of copies of packets. This is depends strongly on the implementation.

Therefore, we only examine our own simulator. Figure 8.9 depicts the use of packets with increasing node numbers. Figure 8.10 shows the number of events. This information can be used to estimate Cubus' future memory consumption. The more packets and events are used, the more memory is needed by the simulator.

8.7. Latency

The previous sections have analyzed the performance of our simulator. In several cases we have compared the results to GTNetS and NS-2. In this section we examine the behavior of the flooding algorithm used in our scenario by measuring the latency of the simulated network.

We measure the *latency* in our scenario, i.e. the average SimTime that is needed until a node receives a flooded packet for the first time. The results of our measurements are depicted in Figure 8.11. The exact numeric values are given in Appendix B.3.

In theory, latency should increase with a growing number of nodes in the same simulation area. The more nodes are located in the same area, the more collisions occur during transmission which increases the average waiting time until a node receives a packet first.

Cubus and GTNetS show similar behavior, varying only in an offset. This offset is depends on the simulation time. GTNetS uses more events and provides more detailed and realistic models for wireless medium and interface. But the NS-2 behavior differs. While the graphs of GTNetS and Cubus seem to converge, the latency in NS-2 grows very slowly in linear order.

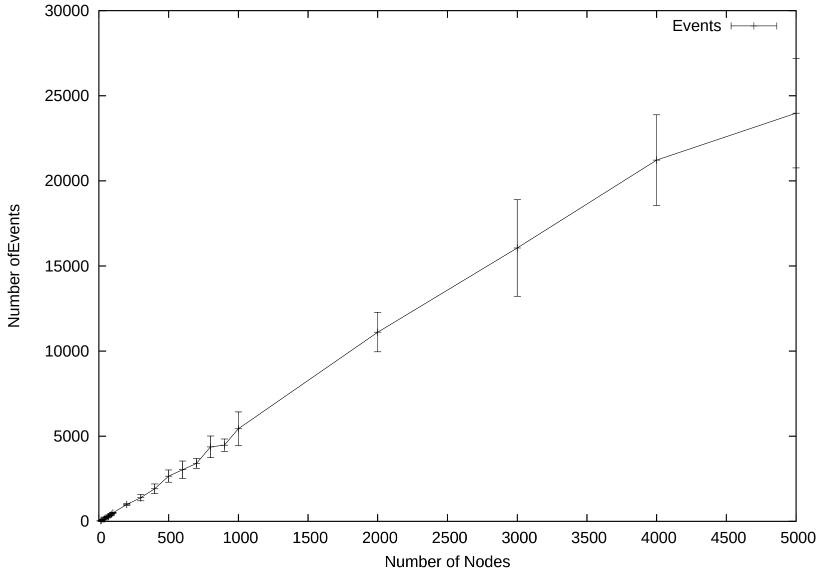

Figure 8.10.: Number of events with increasing number of nodes

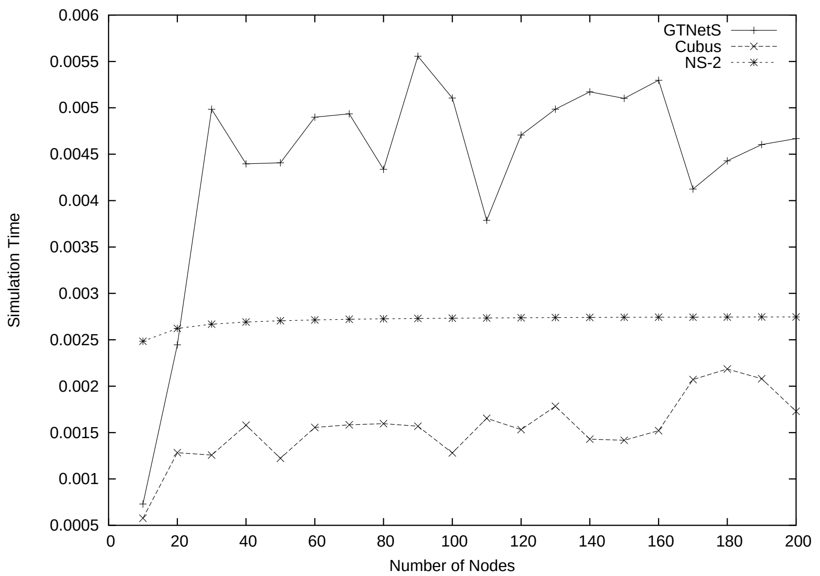

Figure 8.11.: Latency with Cubus, NS-2 and GTNetS

The similar results of GTNetS and Cubus show that our seems to work correctly with regard to the latency. Besides the different behavior, the dimension of the results of both NS-2 and Cubus is almost the same. These results suggest that our prototype implementation is correct.

8.8. Summary

We have simulated a simple scenario using a simple flooding algorithm. A single, randomly chosen node transmits a packet. The simulation runs until all reachable nodes are flooded. Comparisons between three simulators, namely Cubus, GTNetS and NS-2 have shown that Cubus outperforms GTNetS regarding the runtime of the simulator in CPU time. The runtime of Cubus grows in linear order with increasing number of nodes. But, we have to admit that Cubus is not yet finished and lacks some detailed models. Therefore, after implementing detailed models the runtime performance is supposed to worsen a bit. But it should stay in linear order.

The memory usage of Cubus is lower than that of GTNetS. This is due to the flexibility provided by dynamic loading of algorithms. Cubus only loads those components into memory that are really required for the simulation, while GTNetS loads much more components into memory.

The latency measurements in our scenario show similar behavior between Cubus and GTNetS. The behavior of NS-2 differs. However, the magnitude of all three results is similar.

This chapter has provided sufficient information to show that design developed for Cubus is feasible. Although it is hard to compare a simulator which is at its beginning with fully matured ones, we can see from the results that the design is a good basis for further development.

However, it is necessary to provide additional measurements as soon as more detailed models and algorithms are supported.

9. Conclusion

This thesis provided a detailed analysis of the field of network simulation. Its goal was the design and development of a new tool for network simulation and demonstration. The necessity of this was pointed out by giving a detailed overview of the state-of-the-art techniques for network simulation.

Many of the available simulators have been primarily designed for wired networks. Simulations of wireless networks, especially mobile ad hoc networks, are a relatively new topic. Therefore, this is where we focus in this thesis lies.

When examining the related work in Chapter 3, it became obvious, where the weak points of common simulators are, e.g. the lack of modularity and flexibility. But also, the support for mobile ad hoc networks was lacking in some simulators. Also, interactivity was usually neglected. Another problem is the high resource consumption of these simulators.

Based on this, we developed the requirements for our simulator. The requirements are adaptivity, interactivity, modularity and low resource consumption. Additionally, our simulator primarily focusses on mobile ad hoc networks.

In Chapter 4 we have seen the architecture of the Cubus simulator and demonstrator. Its modular and flexible design was pointed out. Both, modularity and flexibility, are important for the design of our simulator.

The design of our simulator, as presented in Chapters 5 and 6, aims to fulfill these requirements, except interactivity. In Chapter 5, we designed the central components of our simulator.

We first focused on the registration of events by using a publish / subscribe mechanism. Thereby, components can subscribe for specific events they are interested in and get informed as soon as that event occurs. In the following, we discussed the treatment of events by the use of handlers and the two central event queues. All these things together offer advantages in simulation speed and provide flexibility through the subscribe mechanism. Furthermore, the flexibility and adaptivity is enhanced by custom parameters. For this a workaround was needed due to the absence of proper reflection in C++.

The design of simulation components is described in Chapter 6. Algorithms were designed, which can be loaded or exchanged during the runtime of a simulation.

In addition, we introduced the modular structure of nodes. Almost arbitrary components can be attached to each node. This allows the construction of nodes suited to individual preferences, e.g. the flexibility in handling the EMUs and the MMUs. They facilitate the definition of the level of detail for each simulation component by the user.

We discussed the communication principles within and between nodes. We evaluated the advantages of these types of communication. This information shows the flexibility, adaptability and modularity of our design is.

Finally, in Chapter 7 we discussed the implementation aspects of our simulator. We have seen which design ideas were implemented during this thesis and which are still open.

The evaluation in Chapter 8 primarily focused on the performance aspects of our simulation tool. From the results presented in that chapter, we have seen that Cubus is a powerful and fast tool for simulating networks. As can be derived from the measurements of memory consumption in Figure 8.8, our tool does indeed consume a low amount of resources.

Although Cubus is far from being a complete and mature simulation tool, the design developed in this thesis provides a solid basis for further development. Some extensions that could be investigated are discussed Chapter 10.

10. Future Work

We have designed and implemented a base system for a discrete event-based network simulator during the course of this thesis. Many important functionalities are available and tested. Still, a good deal of extensions and improvements are required to enable accurate and realistic network simulations. These extensions are beyond the scope of this thesis.

Therefore, we provide some suggestions for improvements and extensions in this chapter. At first, we take a closer look at the central components of the core engine and discuss possible extensions. Finally, we point out extensions for the other simulation components.

10.1. Core Engine

Here are some ideas for extending the central components of the core engine. They range from the insertion of a history functionality to possible parallelization approaches.

10.1.1. History Functionality

A kind of *history* functionality may be desirable. By that means one can go back arbitrary steps in the simulation history. This implies some advantages. One advantage would be the ability to carry out different experiments within the same simulation run. For example the user can go back in the history to a specific point of SimTime. He then may do changes to simulation parameters, like node energy, at this point of time. After that, the simulation can be resumed using the new parameters. The user may be interested in the results of this parameter change compared to the old simulation. This reduces the need of restarting the whole simulation from the beginning.

We will call this process *branching*. Basically, there are two possibilities how branching can be logged. Using the first approach log events are sequentially written to the output medium, e.g. a database or a text file. But branches have to be marked correctly. The second approach tries to model the branching behavior when logging, e.g. by creating a new log text file or a new database table.

This functionality seems to be tempting, but it causes several, partly serious problems and difficulties. Therefore, a detailed analysis or design of it would go beyond the scope of this thesis. We only point out some problems in the following.

State

To understand some problems that arise by the use of the history functionality, we have to define the notion of *state*. A state describes all parameters relevant to a simulation at a specific point of time during that simulation. Among other parameters these parameters consist of the number of nodes, their current protocol stacks and the contents of the central event queues.

State Saving

States have to be saved. Basically, there are two different ways of saving. Typically saving should take place at so called *synchronization points* which are distributed among the simulation queue. First, states can be *fully* saved at each “sync point” by storing them on the harddisk. Storing it in the main memory is not sufficient, since simulation scenarios can grow very large. Also consider the case where every five SimTime seconds a complete scenario is stored.

Secondly, saving of states at “sync points” can be done incrementally, similar to incremental backups in databases [KeEi01]. Using this approach, only changes compared to the last “sync point” are saved. This saves storage space and time, since only a smaller amount of data has to be written to disk.

Problems

Implementing these possibilities causes some problems. The first approach is too storage intensive. Additionally, it slows down the simulation, since saving on harddisk is time consuming. It is comparable to *supsend-to-disk* methods in modern notebooks [ChCB04].

Therefore, only the second approach that uses incremental backups is adequate. *Serializers* and *deserializers* are required for writing objects to disk. Besides the fact, that the developer of new a class has to provide appropriate serializing and deserializing methods, an additional problem arises: A convenient way has to be found for handling global variables of the simulation.

Only when these and additional problems are solved, history functionality can be implemented for our simulator.

10.1.2. Extending Repositories

This thesis has provided the capabilities to dynamically load algorithms during runtime (see Section 6.1). The design of our simulator also supports dynamic loading of other components like network interfaces and media, as described in Chapter 6.

Dynamic loading of these components can be implemented analogously to dynamic loading of algorithms. Additionally, new repositories have to be added to Cubus which store their basic configuration and available parameters (see Section 5.4). This is analogous to algorithm repositories.

If this functionality is provided, the flexibility of our simulator is increased. In the ideal case it should only be necessary to compile the simulator core one time and then load all other required components dynamically during runtime. But it has to be considered that dynamic linking is typically slower than static linking, since some part of the linking process has to be done during runtime. Therefore, a middle course between full flexibility and speed has to be found.

10.1.3. Parallelization

In addition to the previously mentioned possible extensions, *parallelization* or *distribution* is worth discussing. Both may accelerate the simulator. Several possibilities exist for parallelizing our simulator. Some are already included in the design.

First, Cubus uses two independent central event queues. One of them, the system queue, runs in parallel and asynchronously compared to the rest of the simulator. However, this is only a very small approach in parallelization.

Secondly, due to the method-based communication within each node, nodes may run independently from each other. Hence, they are subject to a very fine grained parallel approach. But nodes are very small units and therefore the administration overhead for parallelization is too high. Thus, a coarser-grained approach is required.

Designing a parallel simulation requires the following aspects to be considered: First, efficient synchronization mechanisms are needed to reduce the simulation overhead. Secondly, the simulation needs to be partitioned in a way that all processors almost have the same load. Thirdly, the partitions should be designed to reduce communication between the different processors.

As a possibility for load balancing, *space-filling curves* [AlSe97] can be examined. They allow load balancing in a simple and elegant way.

To reduce the communication overhead, the network has to be partitioned in an appropriate way. One possibility is using *ghost nodes*, as done in GTNetS [RJFA04, Rile03a]. Another one can be done analogously to GloMoSim's partitioning of simulation areas and node aggregation [ZeBG98].

It has to be considered that parallelization is a complex task in mobile networks, due to the mobility of nodes and as a consequence thereof, the changes in the network topology. Nevertheless, parallelization can be worthwhile to investigate.

10.1.4. Emulation Concepts

Emulation concepts are used in a growing number of simulators. They provide a reasonable extension to traditional network simulators. NS-2 also implements an emulation interface [BBEF$^+$99] which allows communication between real-world network nodes and simulated network nodes.

Combining real-world measurements with those of the simulated world seems to be reasonable. Considering a scenario where within our simulator its simulated nodes interact with real-world mobile nodes, e.g. Tablet PCs.

10.2. Simulation Components

This section points out some possible enhancements for components that already exist as well as some possible additions to our simulator. Some of these additions are already under development for another diploma thesis.

10.2.1. Interactivity

As mentioned in Chapter 4, Cubus is supposed to be a network simulator as well as a demonstrator. Therefore, visualization is needed. This can be realized in the form of *user interfaces* (UI), e.g. a graphical user interface.

Using an UI, either a graphical or a text-based one, it is possible to interact with the simulation by changing its simulation parameters. All this is incorporated in the *interactive* *engine*. Possible interactions could be altering the energy levels of nodes or moving nodes to new locations.

10.2.2. Spatial Models

So far, our simulator only supports simulation areas without any obstacles. This complies with an ideal simulation environment where signal path loss is the only crucial parasitic. Only path loss limits the possible transmission range.

But, this model is far from enabling a realistic simulation. In a real-world environment many different and severe parasitics exist. The reason of that are mostly obstacles like buildings. Signals are reflected, scattered or even blocked by these obstacles.

These things are necessary for a realistic simulation and could be modeled by *spatial models*. Spatial models are part of the ongoing thesis mentioned in the previous subsection.

10.2.3. Analysis Capabilities

Log events are used to log specific simulation characteristics. These log events ca be configured almost arbitrarily (see Section 5.6). So far, they are forwarded to the standard output, e.g. printed on the screen.

But this behavior is not sufficient for detailed evaluation of a simulation. Consequently, we need a more complex treatment than simple output. For these purposes an *analysis engine* is desirable. It was introduced into the Cubus architecture in Chapter 4.

It should allow detailed evaluation of specific simulation parameters. Additionally, different application areas need different kinds of outputs. Therefore, the analysis engine should be able make use of different *loggers*, like databases or simple text files. It provides a common interface so that all types of loggers are able to communicate with the simulation engine via this interface. Design and implementation of the analysis engine is currently in progress as well.

10.2.4. Mobility

The main purpose of our simulator is the simulation of mobile ad hoc networks. But mobility is not part of this thesis. Therefore, nodes currently stay on fixed positions in the simulation area. Nodes should incorporate mobility, since this is the only way *mobile* networks can be simulated.

There exist a great variety of mobility models, e.g. the random way point model. They should interact with nodes to simulate their movements. Additionally, an exchange of mobility models during runtime is desirable.

10.2.5. Network Interfaces

This thesis provided a very basic network interface with very limited capabilities. Realistic simulations require the integration of far more detailed network interfaces, that model physical network interfaces more accurately. Additionally, dynamic loading of network interfaces is a good idea. This avoids the need for recompilation after a new network interface is attached to a node or replaces another one in the protocol stack.

10.2.6. Algorithms

We provided two algorithms in this thesis. The first one is the MAC layer algorithm IEEE 802.11. The second one is the simple flooding algorithm (see Chapter 8) that was implemented for evaluation purposes.

Simulations require many more algorithms. Possible ideas are IEEE 802.3 for wired networks, IPv6, TCP variants and routing algorithms like AODV. Most of them can be ported from GTNetS or other similar simulators.

10.2.7. Propagation Models

We have implemented a very basic propagation model during this thesis, s simple free space path loss propagation model. Different and more accurate models are needed for detailed, realistic simulation, e.g. two ray ground. The more accurate they are, the more realistic the simulation will be.

A. Configuration and Scenario Files

This chapter closely examines configuration and scenario files. First, we describe the LALR(1) grammar used for generating the parser. Flex and Yacc have been used to generate the parser. After this, we show some small examples which illustrate the use of configuration and scenario files.

A.1. Parser Grammar

We describe the parser grammar in the following. It shows the current state of the parser. It needs to be extended for handling additional components like mobility models.

data	→	scenario | algorithm
scenario	→	inf med nodes conIfAlg links event
inf	→	cinterfaces { dynparam };
med	→	cmedium { dynparam };
links	→	link
		| link links
nodes	→	node | node nodes
node	→	cnode { parameter stack };
conIfAlg	→	conif
conif	→	con
		| con conif
parameter	→	pe
		| pe parameter
pe	→	VARIABLE = DOUBLE;
		| VARIABLE = VARIABLE;
		| VARIABLE = INTEGER;
stack	→	cprotocolStack { definition connections };
definition	→	dynparam
connections	→	con
		| con connections
con	→	cregister (VARIABLE , VARIABLE);
link	→	sharedLink { linking };
linking	→	lp
		| lp linking
lp	→	VARIABLE connect VARIABLE (VARIABLE);
dynparam	→	dp
		| dp dynparam
dp	→	VARIABLE VARIABLE VARIABLE;
event	→	eventp
		| eventp event
eventp	→	cevent { parameter };
algorithm	→	parameter propdef
propdef	→	pf

		| pf propdef
pf	→	property { parameter };

To understand the grammar, we present some examples for configuration files and scenarios in the following sections.

A.2. Algorithm Configuration Files

As we have seen in Section 5.4, we need configuration files for every dynamically loadable object. Since we only support algorithms as dynamically loadable objects, we only support *algorithm configuration files*. Each algorithm requires one file. Four parameters are mandatory: the name of the algorithm, its type, its location and the local log level. All other parameters are optional. A simple configuration file may look like.

```
name = "SimpleFlooding";
type = "../examples/SimpleFlooding.so";
location = "../examples/SimpleFlooding.so";
loglevel = 5;

property{
   name = customParam1;
   min = 5;
   max = 25;
   step = 1;
   editable = true;
};
```

An arbitrary number of custom parameters are supported. They are configured with the `property` construct.

A.3. Scenario Files

In this section we provide a complete, simple scenario. It models the scenario presented in Section 8.3.1. We define two nodes in this scenario.

```
cinterfaces {
   Interface loc nd0;
   Interface loc nd1;
};

cmedium {
   Wlan loc med1;
};

cnode {
   name = node0;
   memory = 5000000;
   energy = 230;
   EMU = simple;
   MMU = simple;
```

```
   cpu = 1200000000;
   architecture = 32;
   cprotocolStack{
      Application "../examples/FloodingApp.so" c0;
      Algorithm "../examples/libieee802.11alg.so" a0;
      Algorithm "../examples/SimpleFlooding.so" b0;
      cregister(a0, b0);
      cregister(b0, c0);
   };
};

cnode {
   name = node1;
   memory = 5000000;
   energy = 230;
   EMU = simple;
   MMU = simple;
   cpu = 1200000000;
   architecture = 32:
   cprotocolStack{
      Application "../examples/FloodingApp.so" c1;
      Algorithm "../examples/libieee802.11alg.so" a1;
      Algorithm "../examples/SimpleFlooding.so" b1;
      cregister(a1, b1);
      cregister(b1, c1);
   };
};

cregister(nd0, a0);
cregister(nd1, a1);

sharedLink{
   med1 -> node0(nd0);
   med1 -> node1(nd1);
};

cevent{
   type = ApplicationSend;
   source = c1;
   time = 5.0;
};
```

We create two nodes, each with five megabytes of memory, an initial energy level of 230 mW, as well as a CPU of 1.2 GHz. The architecture is based on 32 bit. Both nodes are interconnected via a wireless medium. At SimTime 5.0 `node1` starts to transmit a packet via SimpleFlooding.

B. Numeric simulation results

This chapter provides the numeric values of the measurements that have been described in Chapter 8. We first present the values of Cubus and then thereafter GTNetS.

B.1. Cubus

The columns are defined in the following way. The first column describes the number of nodes in the simulation run. The second column shows the number of the current run. *CPU* and *Setup* denote the simulation runtime in CPU time and the setup time respectively. The next two columns, *Mem1* and *Mem2*, show the memory usage before and after the scenario was built. The last two ones give the number of packets and events during the specific run.

Nodes	Run	CPU	Setup	Mem1	Mem2	Packets	Events
10	1	0.02	0.02	3	4	54	33
10	2	0.01	0.04	3	4	46	25
10	3	0.01	0.01	3	4	62	37
10	4	0.02	0.01	3	4	22	13
10	5	0.03	0.02	3	4	56	33
20	1	0.05	0.04	4	4	146	79
20	2	0.04	0.04	4	4	108	59
20	3	0.03	0.03	4	4	132	72
20	4	0.02	0.05	4	4	82	45
20	5	0.05	0.04	4	4	132	73
30	1	0.04	0.05	3	4	182	98
30	2	0.07	0.03	4	4	222	120
30	3	0.07	0.04	4	4	182	96
30	4	0.09	0.02	4	4	268	114
30	5	0.07	0.04	4	5	228	121
40	1	0.14	0.06	4	4	306	160
40	2	0.1	0.06	4	4	276	145
40	3	0.09	0.05	4	4	264	140
40	4	0.08	0.02	3	4	294	154
40	5	0.11	0.06	4	5	280	146
50	1	0.12	0.1	4	4	406	212
50	2	0.14	0.04	4	4	468	245
50	3	0.13	0.07	4	5	420	219
50	4	0.17	0.06	3	4	438	228
50	5	0.09	0.06	4	4	350	183
60	1	0.22	0.08	4	4	528	274
60	2	0.20	0.05	4	5	598	308
60	3	0.18	0.04	4	5	568	294
60	4	0.14	0.10	3	4	450	232
60	5	0.17	0.05	3	4	464	240

70	1	0.21	0.11	4	5	506	259
70	2	0.21	0.09	3	4	550	282
70	3	0.18	0.06	3	4	512	263
70	4	0.22	0.08	3	4	580	299
70	5	0.17	0.10	3	4	540	278
80	1	0.23	0.05	4	5	740	379
80	2	0.27	0.12	4	5	734	376
80	3	0.23	0.12	4	4	634	325
80	4	0.27	0.10	3	4	706	362
80	5	0.20	0.09	4	5	646	331
90	1	0.27	0.10	4	5	826	423
90	2	0.28	0.15	4	6	918	469
90	3	0.25	0.10	4	5	636	326
90	4	0.31	0.08	4	5	846	432
90	5	0.29	0.14	4	5	818	420
100	1	0.34	0.15	4	5	934	478
100	2	0.35	0.10	4	5	1008	516
100	3	0.33	0.13	3	4	1012	517
100	4	0.28	0.11	5	6	850	435
100	5	0.32	0.12	4	5	1038	531
200	1	0.6	0.22	3	6	1896	958
200	2	0.73	0.22	3	6	2088	1056
200	3	0.59	0.29	4	6	1838	930
200	4	0.66	0.3	4	7	1992	1006
200	5	0.7	0.22	5	7	1974	999
300	1	1.07	0.36	4	7	3314	1671
300	2	0.86	0.41	3	6	2632	1325
300	3	0.94	0.47	3	6	2872	1446
300	4	0.85	0.34	4	7	2340	1179
300	5	0.91	0.36	4	7	2650	1334
400	1	1.42	0.46	4	8	4240	2132
400	2	0.83	0.49	4	8	3076	1546
400	3	1.38	0.48	3	8	4384	2205
400	4	1.27	0.51	4	8	3958	1991
400	5	1.05	0.47	4	8	3362	1691
500	1	1.86	0.66	3	9	5574	2860
500	2	1.52	0.67	4	10	4556	2289
500	3	1.5	0.57	3	9	5230	2627
500	4	1.99	0.67	3	9	6284	3157
500	5	1.55	0.71	3	9	4676	2348
600	1	1.67	0.77	4	10	5486	2753
600	2	2.11	0.79	3	10	6902	3467
600	3	1.85	0.72	4	10	5626	2823
600	4	1.59	0.83	3	10	4874	2448
600	5	2.43	0.72	4	10	7308	3668
700	1	2.03	0.85	4	11	6316	3169
700	2	2.12	0.92	4	11	6492	3258
700	3	2.48	0.85	4	11	7148	3585
700	4	2.00	0.85	4	11	6344	3182
700	5	2.36	0.98	4	11	7614	3820
800	1	2.72	1.03	3	12	8582	4303

800	2	2.49	0.91	4	13	7866	3943
800	3	3.5	1.02	4	12	10072	5050
800	4	3.45	1.13	4	12	9930	4979
800	5	2.46	1.17	4	13	7172	3595
900	1	2.6	1.19	4	14	9042	4535
900	2	2.8	1.15	4	14	9054	4540
900	3	3.37	1.2	4	13	9954	4992
900	4	2.92	1.31	4	13	8606	4314
900	5	2.59	1.09	4	13	7978	3999
1000	1	3.19	1.38	4	14	9100	4563
1000	2	4.4	1.47	4	14	13420	6729
1000	3	3.2	1.48	3	14	9420	4721
1000	4	3.18	1.2	3	14	9760	4892
1000	5	4.22	1.4	3	13	12520	6278
2000	1	6.79	2.77	3	23	19830	9927
2000	2	8.05	2.83	4	24	23776	11900
2000	3	6.72	2.73	4	25	21262	10645
2000	4	6.9	2.74	3	24	20746	10384
2000	5	8.35	2.64	4	24	25406	12716
3000	1	9.52	4.12	3	34	29936	14979
3000	2	9.13	3.97	3	34	29094	14558
3000	3	10.03	3.94	4	34	30254	15138
3000	4	14.27	4.06	3	34	42176	21106
3000	5	9.77	4.05	3	34	28988	14506
4000	1	14.7	5.58	3	44	45032	22531
4000	2	15.38	5.57	3	44	47682	23855
4000	3	12.52	5.4	4	45	38966	19494
4000	4	10.82	5.29	4	45	34856	17438
4000	5	14.8	5.45	3	44	45520	22775
5000	1	11.92	5.75	4	55	38288	19154
5000	2	15.56	7.01	4	55	48160	24092
5000	3	14.76	6.86	4	55	46530	23277
5000	4	17.91	6.91	4	54	55774	27901
5000	5	16.46	6.52	4	55	50912	25468

Table B.1.: Numeric values for Cubus

B.2. GTNetS

The columns in this table are arranged in the same way as in Section B.1.

Nodes	Run	CPU	Setup	Mem1	Mem2	Packets	Events
10	1	0.01	0.05	16	16	37	52
10	2	0.01	0.07	17	17	61	80
10	3	0.001	0.08	17	17	14	27
10	4	0.001	0.07	16	16	37	54
10	5	0.01	0.07	16	16	63	91
20	1	0.03	0.1	16	16	252	323
20	2	0.03	0.06	16	16	241	309
20	3	0.03	0.07	17	17	234	287

20	4	0.01	0.06	17	17	153	205
20	5	0.03	0.07	16	16	167	223
30	1	0.07	0.06	17	17	430	546
30	2	0.07	0.07	16	16	437	550
30	3	0.05	0.07	16	16	505	621
30	4	0.05	0.07	16	16	311	404
30	5	0.06	0.06	17	17	464	565
40	1	0.08	0.07	16	16	703	864
40	2	0.11	0.06	16	16	751	932
40	3	0.11	0.07	16	16	742	951
40	4	0.09	0.07	17	17	774	933
40	5	0.11	0.06	16	16	773	970
50	1	0.11	0.05	15	15	1067	1314
50	2	0.10	0.04	16	16	1190	1451
50	3	0.13	0.07	16	16	1128	1356
50	4	0.14	0.05	16	16	1064	1300
50	5	0.12	0.06	16	16	1322	1575
60	1	0.14	0.06	16	16	1477	1828
60	2	0.17	0.04	15	15	1715	2067
60	3	0.17	0.07	15	15	1632	1931
60	4	0.18	0.07	16	16	1406	1710
60	5	0.10	0.06	16	16	1857	2241
70	1	0.25	0.04	16	16	2212	2732
70	2	0.21	0.04	15	15	2242	2636
70	3	0.27	0.06	16	16	2207	2644
70	4	0.22	0.06	15	15	1895	2372
70	5	0.20	0.05	15	15	1886	2351
80	1	0.29	0.05	16	16	2985	3573
80	2	0.28	0.05	15	15	2944	3466
80	3	0.29	0.04	15	15	3043	3694
80	4	0.26	0.06	15	15	2843	3436
80	5	0.23	0.06	15	15	2664	3328
90	1	0.29	0.06	16	16	3676	4446
90	2	0.37	0.07	15	15	3581	4390
90	3	0.38	0.07	15	15	3699	4387
90	4	0.36	0.07	15	15	3678	4329
90	5	0.39	0.07	15	15	3646	4283
100	1	0.47	0.06	15	16	4533	5438
100	2	0.46	0.05	15	15	4462	5334
100	3	0.41	0.08	16	16	4461	5363
100	4	0.41	0.07	15	15	4499	5455
100	5	0.44	0.04	15	15	4240	5065
200	1	1.72	0.05	15	16	17437	20346
200	2	1.62	0.06	15	15	17995	21376
200	3	1.78	0.04	15	15	17169	19912
200	4	1.64	0.07	15	16	16324	19038
200	5	1.56	0.06	15	16	15969	18969
300	1	4.16	0.08	15	15	39960	46316
300	2	3.75	0.09	14	15	37186	43240
300	3	3.71	0.08	14	15	35966	41762
300	4	3.84	0.09	16	16	37745	43346

300	5	4.04	0.08	15	16	37425	43269
400	1	7.19	0.07	15	15	68277	77809
400	2	6.68	0.08	15	15	68011	78034
400	3	7.50	0.07	16	16	88933	76611
400	4	6.98	0.09	15	16	68650	78599
400	5	6.72	0.08	16	16	66498	76015
500	1	10.38	0.1	15	16	103126	117801
500	2	10.59	0.09	15	15	99884	113717
500	3	10.63	0.12	15	16	99943	114055
500	4	10.76	0.11	15	16	102810	118213
500	5	10.83	0.09	16	17	105696	120425
1000	1	41.44	0.15	15	32	410891	458237
1000	2	40.91	0.15	15	31	416622	468139

Table B.2.: Numeric values for GTNetS

B.3. Latency Measurements

This section presents the results of our latency measurements. We present the values of NS-2, GTNetS and Cubus. The values are given in SimTime.

Nodes	Run	NS-2	GTNetS	Cubus
10	1	0.00248426	0.000918684	0.000131658
10	2	0.00248406	0.000340336	0.000460975
10	3	0.0024843	0.000277682	0.000856124
10	4	0.00248408	0.00159388	0.000839584
10	5	0.00248418	0.000520049	0.000592745
20	1	0.00262223	0.000358053	0.00136172
20	2	0.00262214	0.00305748	0.00237819
20	3	0.00262229	0.00490843	0.000790268
20	4	0.00262236	0.00220375	0.000776166
20	5	0.0026221	0.00169871	0.00110966
30	1	0.00266811	0.00514007	0.000790272
30	2	0.00266806	0.00348796	0.00116442
30	3	0.00266827	0.00524237	0.00139958
30	4	0.00266831	0.00607404	0.00156061
30	5	0.00266817	0.00497622	0.00136989
40	1	0.00269123	0.00614921	0.00115125
40	2	0.00269124	0.00529531	0.00212766
40	3	0.00269138	0.00313534	0.00134506
40	4	0.00269098	0.00439692	0.00144668
40	5	0.00269108	0.00300837	0.00183008
50	1	0.00270481	0.00280699	0.00104936
50	2	0.00270496	0.00475353	0.00126611
50	3	0.00270495	0.00547125	0.00105407
50	4	0.00270494	0.00486377	0.00137906
50	5	0.00270495	0.00414494	0.0013661
60	1	0.00271418	0.00474452	0.00110288
60	2	0.0027143	0.00583339	0.00135605
60	3	0.00271442	0.00469941	0.0011142
60	4	0.00271423	0.00429413	0.00235091

60	5	0.0027144	0.00492314	0.00184815
70	1	0.00272084	0.00699531	0.00216274
70	2	0.0027207	0.00449498	0.00128998
70	3	0.00272086	0.00425425	0.001616
70	4	0.00272062	0.00417144	0.00158226
70	5	0.00272064	0.00476195	0.00126837
80	1	0.00272564	0.00411383	0.0010098
80	2	0.00272558	0.00411491	0.00204369
80	3	0.00272584	0.00520767	0.00151734
80	4	0.00272555	0.00385999	0.00142138
80	5	0.00272585	0.00438777	0.0019843
90	1	0.00272939	0.00499052	0.00147428
90	2	0.00272958	0.00526849	0.00108458
90	3	0.00272956	0.00658729	0.00176105
90	4	0.00272967	0.00471977	0.00208804
90	5	0.00272967	0.00621686	0.00143662
100	1	0.00273292	0.00431258	0.00101969
100	2	0.00273266	0.00515965	0.00119085
100	3	0.00273261	0.00556101	0.00150898
100	4	0.00273265	0.00529149	0.00106299
100	5	0.0027327	0.00520272	0.00161933
110	1	0.00273515	0.00402797	0.00198721
110	2	0.00273512	0.00390812	0.00204603
110	3	0.00273525	0.00339208	0.00106785
110	4	0.00273501	0.00414749	0.0012804
110	5	0.00273541	0.00346004	0.00188648
120	1	0.00273739	0.00395034	0.00176051
120	2	0.00273743	0.00367646	0.0019523
120	3	0.00273715	0.00366318	0.00149213
120	4	0.00273735	0.00642666	0.00135095
120	5	0.00273741	0.00581485	0.00110098
130	1	0.00273901	0.00437919	0.00106999
130	2	0.00273912	0.00472373	0.00187743
130	3	0.00273931	0.00760297	0.00266932
130	4	0.00273904	0.00443128	0.001846
130	5	0.0027394	0.00379089	0.00145272
140	1	0.0027406	0.00637214	0.00134116
140	2	0.00274037	0.0045233	0.00158411
140	3	0.00274059	0.00601958	0.00127899
140	4	0.00274051	0.00308313	0.00147562
140	5	0.00274052	0.00586554	0.00146289
150	1	0.0027417	0.00551937	0.00144502
150	2	0.00274188	0.00469675	0.00116635
150	3	0.00274204	0.00297061	0.00144829
150	4	0.00274187	0.00644706	0.00153137
150	5	0.00274192	0.00587503	0.00149521
160	1	0.0027431	0.00689537	0.00115138
160	2	0.00274304	0.005259	0.00170845
160	3	0.00274328	0.0037299	0.00122793
160	4	0.00274283	0.00502498	0.00143432
160	5	0.00274301	0.00557625	0.00207986
170	1	0.00274414	0.00390087	0.00213066

170	2	0.00274413	0.0041486	0.00228042
170	3	0.002744	0.00446204	0.00196771
170	4	0.00274383	0.00342162	0.00207605
170	5	0.00274383	0.00468908	0.00190316
180	1	0.00274486	0.0040179	0.00216438
180	2	0.00274485	0.00439802	0.00217728
180	3	0.00274482	0.00459173	0.00220691
180	4	0.00274475	0.00494015	0.00244187
180	5	0.00274496	0.00419445	0.00193453
190	1	0.00274581	0.00447885	0.00260659
190	2	0.00274576	0.00586993	0.00203178
190	3	0.00274562	0.00522471	0.00156852
190	4	0.00274582	0.00393095	0.00222986
190	5	0.00274601	0.00351711	0.00196671
200	1	0.00274624	0.0048752	0.00121221
200	2	0.00274625	0.00472796	0.00168048
200	3	0.00274661	0.00370982	0.00180926
200	4	0.00274654	0.00407889	0.00212062
200	5	0.00274636	0.00595195	0.00182243

Table B.3.: Numeric values of latency measurements in SimTime

Bibliography

[Abra85] N. Abramson. Development of the ALOHANET. *IEEE Trans. on Information Theory* Volume IT-31, March 1985, p. 119–123.

[AhSU88] A. V. Aho, R. Sethi and J. D. Ullman. *Compilerbau.* Addison-Wesley GmbH. 3. unveränderter Nachdruck, 1988.

[AlSe97] S. Aluru and F. Sevilgen. Parallel domain decomposition and load balancing using space-filling curves, 1997.

[AWSC02] I. Akyildiz, S. Weilian, Y. Sankarasubramaniam and E. Cayirci. A survey on sensor networks. *Communications Magazine* Volume 40, August 2002, p. 102–114.

[Bake02] F. Baker. An outsider's view of MANET. *Internet Engineering Task Force*, March 2002.

[Ball96] P. D. Ball. Introduction to discrete event simulation. In *Proceedings of 2nd DYCOMANS workshop on 'Management and Control: Tools in action', Algarve Portugal*, May 1996, p. 367–376.

[BaSB03] F. Baumgartner, M. Scheidegger and T. Braun. Enhancing Discrete Event Network Simulators with Analytical Network Cloud Models. In *International Workshop Inter-domain Performance and Simulation, Salzburg*, 2003, p. 21–30.

[BBEF+99] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu and D. Zappala. Improving Simulation for Network Research. Technical Report 99-702b, University of Southern California, March 1999. revised September 1999, to appear in IEEE Computer.

[BDSZ94] V. Bharghavan, A. Demers, S. Shenker and L. Zhang. MACAW – A Media Access Protocol for Wireless LANs. In *Proceedings SIGCOMM '94 Conference.* ACM, 1994, p. 212–225.

[BEFF+00] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu and H. Yu. Advances in Network Simulation. *Computer* 33(5), 2000, p. 59–67.

[Bind04] R. Bindal. Wireless Internet Post Office. Technical Report, Indian Institute of Technology Delhi, 2004.

[BoEl95] D. Bolier and A. Eliens. SIM: a C++ library for Discrete Event Simulation, 1995. Vrije Universiteit, Department of Mathematics and Computer Science, Amsterdam, The Netherlands.

[BTAT+99] L. Bajaj, M. Takai, R. Ahjua, K. Tang, R. Bagrodia and M. Gerla. GloMoSim: A scalable network simulation environment. Technical Report, UCLA Computer Science Department, 1999.

[CaSS02] D. Cavin, Y. Sasson and A. Schiper. On the Accuracy of MANET Simulators. In *Proceedings of the Workshop on Principles of Mobile Computing (POMC'02)*. ACM, October 2002, p. 38–43.

[ChCB04] F. Chabaud, N. Cunningham and B. Blackham. Software Suspend Howto. http://softwaresuspend.berlios.de/, August 2004.

[ChGr00] I. Chihaia and T. Gross. Effectiveness of simple memory models for performance prediction, 2000. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'04).

[ClCS02] B. Clarke, A. Czezowski and P. Strazdins. Implementation Aspects of Sparc V9 Complete Machine Simulator, 2002.

[CMU 98] CMU Monarch. The CMU Monarch Project's Wireless and Mobility Extensions to NS, 1998.

[Cobb00] A. Cobbs. All About Netgraph. http://ezine.daemonnews.org/200003/, March 2000.

[Crai96] D. C. Craig. Extensible Hierarchical Object-Oriented Logic Simulation with an Adaptable Graphical User Interface. Master thesis, Memorial University of Newfoundland, 1996.

[DoFN93] M. Doyle, T. Fuller and J. Newman. Modeling of the galvanostatic charge and discharge of the lithium/polymer/insertion cell. *Journal Electrochemical Society*, 1993, p. 140(6).

[Fall99] K. Fall. Network Emulation in the VINT/NS Simulator. *Proceedings of the fourth IEEE Symposium on Computers and Communications*, 1999.

[FaVVp04] K. Fall, K. Varadhan and the VINT project. The ns manual, July 2004.

[Fish01] G. S. Fishman. *Discrete-Event Simulation: modeling, programming and analysis*. Springer-Verlag New Yok, Inc. 2001.

[GHJV96] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Entwurfsmuster*. Addison-Wesley-Longman. 5., korrigierter Nachdruck, 1996.

[GrJa98] D. Grune and C. Jacobs. *Parsing Techniques – A Practical Guide*. Vrije Universiteit Amsterdam. 1998.

[GrKL04] I. Gruber, O. Knauf and H. Li. Performance of Ad Hoc Routing Protocols in Urban Environments. In *The Fifth European Wireless Conference*, February 2004.

[HaTi03] M. Handy and D. Timmermann. Simulation of mobile wireless networks with accurate modelling of non-linear battery effects. In *Proceedings of International Conference on Applied Simulation and Modelling*, 2003, p. 532–537.

[HeRo02] D. Herrscher and K. Rothermel. A Dynamic Network Scenario Emulation Tool. In *Proceedings of the 11th International Conference on Computer Communications and Networks (ICCCN 2002)*, October 2002, p. 262–267.

[HuPe91] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering* 17(1), 1991, p. 64–76.

[IEEE99a] IEEE Computer Society. IEEE Std. 802.11-1999 Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1999.

[IEEE99b] IEEE Computer Society. IEEE Std. 802.11b-1999 Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-Speed Physical Layer Extension in the 2.4 GHz Band, 1999.

[IEEE02] IEEE Computer Society. IEEE Std. 802.3-2002 802.3 Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access methods and physical layer specifications, 2002.

[John99] D. B. Johnson. Validation of Wireless and Mobile Network Models and Simulation. In *Proceedings of the DARPA/NIST Workshop on Validation of Large-Scale Network Models and Simulation*, May 1999.

[Karn90] P. Karn. MACA – A new Channel Access Protocol for Packet Radio. In *ARRL/CRRL Amateur Radio Ninth Computer Networking Conference*, 1990, p. 134–140.

[KeEi01] A. Kemper and A. Eickler. *Datenbanksysteme – Eine Einführung*. Oldenbourg. 4. Auflage, 2001.

[Kesh88] S. Keshav. REAL: A Network Simulator. Technical Report CSD-88-472, University of California at Berkeley, 1988.

[KGMS+01] M. Kojo, A. Gurtov, J. Mannner, P. Sarolahti, T. Alanko and K. Raatikainen. Seawind: a wireless network emulator, 2001.

[KlTo75] L. Kleinrock and F. Tobagi. Packet Switching in Radio Channels: Part II – The Hidden Terminal Problem in Carrier Sense Multiple-Access and the Busy-Tone Solution. *IEEE Transactions on Communications* Volume COM-23, No.12, December 1975, p. 1417–1433.

[Kniz04] K. Knizhnik. C++ Reflection Package. http://www.garret.ru/˜knizhnik/-cppreflection/, July 2004.

[LeMB95] J. R. Levine, T. Mason and D. Brown. *lex & yacc*. O'Reilly & Associates, Inc. 2nd edition, 1995.

[Lemm02] J. J. Lemmon. Wireless Link Statistical Bit Error. *NTIA Report 02-394*, June 2002.

[LiJo03] T. Li and L. K. John. Run-Time Modeling and Estimation of Operating System Power Consumption. In *Proceedings of SIGMETRICS'03*, 2003.

[Luce04] Lucent Technologies. WaveLAN. http://www.wavelan.com/, 2004.

[MaBJ99] D. Maltz, J. Broch and D. Johnson. Experiences designing and building a multi-hop wireless ad hoc network testbed, 1999.

[MaBJ00] D. A. Maltz, J. Broch and D. B. Johnson. Quantitative lessons from a full-scale multi-hop wireless ad hoc network testbed. In *Proceedings of WCNC 2000*, September 2000.

[MaOb04] C. B. Margi and K. Obraczka. An Energy Model for Evaluation of Protocols for Power-Constrained Networks. Technical Report, UCSC, 2004.

[Maue03] W. Mauerer. *Linux Kernelarchitektur*. Carl Hanser Verlag München Wien. 1. Auflage, 2003.

[McCl98] G. McCluskey. Using Java Reflection. http://java.sun.com/developer/, January 1998.

[McFl97] S. McCanne and S. Floyd. The LNBL Network Simulator, 1997.

[NaGr03] V. Naoumov and T. Gross. Simulation of large ad hoc networks. In *Proceedings of the 6th international workshop on Modeling analysis and simulation of wireless and mobile systems*, 2003, p. 50–57.

[Nort00] J. Norton. Dynamic Class Loading for C++ on Linux. *Linux Journal*, May 2000.

[OPNE04] OPNET Technologies Inc. OPNET Modeler. http://www.opnet.com/, August 2004.

[Orfa04] S. J. Orfanidis. *Electromagnetic Waves and antennas.* Rutgers University. July 2004.

[Oual02] S. Oualline. *Practical C++ Programming.* O'Reilly & Associates. 2nd edition, 2002.

[Page91] B. Page. *Diskrete Simulation. Eine Einführung mit Modula-2.* Springer-Lehrbuch. 1991.

[Perk01] C. E. Perkins. *Ad Hoc Networking.* Addison-Wesley, New York. 2001.

[Perl03] M. Perlick. C++ Reflection and Service Library. http://reflection.sf.net, 2003.

[PiSM95] J. Pierce, M. D. Smith and T. Mudge. Instrumentation tools. In *Fast simulation of computer architectures.* Kluwer Academic Publishers, Boston, MA, 1995.

[PrKP02] P. Prinz and U. Kirch-Prinz. *C++: Lernen und professionell anwenden.* mitp-Verlag, Bonn. 2., vollständig überarbeitete Auflage, 2002.

[RaVr01] D. N. Rakhmatov and S. B. Vrudhula. An analytical high-level battery model for use in energy management of portable electronic systems. In *Proceedings of International Conference on Computer Aided Design (ICCAD'01)*, 2001.

[Rile03a] G. F. Riley. Large-scale network simulations with GTNetS. In *Proceedings of the 2003 Winter Simulation Conference*, 2003.

[Rile03b] G. F. Riley. The Georgia Tech Network Simulator. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, 2003, p. 5–12.

[Rizz97] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review* Volume 27, 1997, p. 31–41.

[RJFA04] G. F. Riley, T. M. Jaafar, R. M. Fuijimoto and M. H. Ammar. Space–Parallel Network Simulations using Ghosts. In *Proceedings of the eighteenth workshop on Parallel and distributed simulation*, 2004, p. 170–177.

[Robe72] L. G. Roberts. Extensions of Packet Communication Technology to a Hand Held Personal Terminal. In *Proceedings of the Spring Joint Computer Conference*, 1972, p. 295–298.

[Scal04] Scalable Network Technologies Inc. The QualNet user manual. http://www.scalable-networks.com/, 2004.

[Schi00] J. Schiller. *Mobile communications.* Pearson Education, London. 2000.

[Sili04] Silicon Graphics, Inc. Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/, September 2004.

[Tane95] A. S. Tanenbaum. *Moderne Betriebssysteme*. Carl Hanser Verlag München Wien. 2., verbesserte Auflage, 1995.

[Tane03] A. S. Tanenbaum. *Computernetzwerke*. Pearson Education Deutschland GmbH. 4. Auflage, 2003.

[TecO04] TecO. SmartIts Hardware Platform. http://smart-its.teco.edu/, September 2004.

[UhMu97] R. A. Uhlig and T. N. Mudge. Trace-Driven Memory Simulation: A Survey. *ACM Computing Surveys* 29(2), 1997, p. 128–170.

[Varg01] A. Varga. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, June 2001.

[vDor04] W. van Dorst. BogoMips mini-Howto. *The Linux Documentation Project*, August 2004.

[Voll00] D. Vollmann. Metaclasses and Reflection in C++. http://www.vollmann.com/, 2000.

[ZeBG98] X. Zeng, R. Bagrodia and M. Gerla. GloMoSim: a Library for Parallel Simulation of Large-scale Wireless Networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulations – PADS '98*, May 1998.

[ZhLi02] Y. Zhang and W. Li. An integrated Environment for testing mobile ad-hoc networks. In *Proceedings of Mobihoc 2002, Lausanne, Switzerland*, 2002.