Institut für Parallele und Verteilte Systeme

Abteilung Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70565 Stuttgart

Diplomarbeit Nr. 2206

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

Serena Fritsch

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer:	Dipl. Inf. Susanne Bürklen
	Dipl. Inf. Dominique Dudkowski
	Dipl. Inf. Jörg Hähner
	Dr. rer. nat. Pedro José Marrón
Begonnen am:	24. Mai 2004
Beendet am:	23. November 2004
CR-Klasssifikation:	C.2.1, C.4, I.6.1, I.6.8

Abstract

Mobile ad hoc networks (MANETs) are anticipated to play an important role in the future of networking, making the performance analysis of algorithms and protocols for MANETs an important topic.

The mobility of the nodes and the characteristics of these wireless networks, such as a frequently changing topology, unreliable communication links, and a limited transmission range, require the development of algorithms and protocols performing efficiently under such circumstances.

Measuring the performance of algorithms and protocols for MANETs is complicated by the costs associated with conducting a real-world experiment and the conditions of such an experiment not being easily repeatable. Therefore, as with wired networks, network simulation is common practice for evaluating the performance of such a system.

Traditional network simulators designed for simulating wired networks lack the support necessary for mobile networks, like modeling node movement and spatial constraints (e.g. propagation obstacles). In the past, some of these network simulators have been adapted or newly developed to also support wireless networks. However, few of these support the modeling of spacial constraints. Also, state-of-the-art network simulators generally have a high resource consumption and lack user interactivity, modularity, and adaptability. Therefore, we have developed Cubus, a network simulator fully capable of simulating MANETs and supporting the aforementioned additional features.

The development of the system was split across two theses: In the first thesis, the core system was developed. The support of node movement, modeling of spacial constraints, and user interactivity is developed in this thesis.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Mobile ad hoc networks (MANETs) are anticipated to play an important role in future networks. They can be operated in the absence of an existing infrastructure, such as the Internet, and allow to support rescue missions or coordination of forces in a disaster area. A MANET consists of a number of mobile nodes, such as PDAs, cell phones, or specialized devices, which communicate spontaneously through some wireless communication technology, e.g. Bluetooth or IEEE 802.11. It is desirable to adapt the algorithms and protocols used in existing static networks, since they are well-tested and already deployed in the physical world. However, MANETS differ from their wired counterparts in many issues.

The topology of MANETs frequently changes due to the mobility of the nodes: Nodes can move out of each others transmission range, leading to unreliable communication links between nodes. Also, the energy available for the nodes is often restricted due to the limited life span of batteries. In order to reliably deliver data to a receiving node, these aspects have to be considered when designing algorithms and protocols (e.g. routing protocols) for MANETs.

Additional difficulties are encountered when evaluating these protocols and algorithms on real systems: First, in order to reproduce an experiment, repeatable conditions have to be ensured. For MANETs, these conditions are influenced by environmental constraints and user mobility, making this goal hard to achieve. Also, the number of mobile nodes required, along with the staff to conduct the experiment, often introduces prohibitive costs. Simulation tools, such as network simulators, model the whole system in software, providing an inexpensive alternative for evaluating protocols and algorithms.

Since network simulators have originated from the field of static networks, they have to be adapted to fit the needs of wireless networks, among them are support for node movement and modeling of spatial constraints, like obstacles which attenuate signal propagation. When running a simulation, the user is often interested in the internals of the system. Therefore, the system has to provide visualization capabilities, and a way to modify system parameters while running the simulation. Additionally, a modular system architecture is desirable to allow the easy exchange of algorithms and protocols during runtime.

Existing network simulators, like the *Network Simulator 2 (NS-2)* and the *Georgia Tech Network Simulator (GTNetS)*, lack sufficient support for MANETs. They do not provide adaptivity, since the user is not able to change algorithms and protocols during the running simulation. Also, they lack the support for user interactivity to change parameters of algorithms during a simulation run. To overcome these drawbacks, we have developed Cubus, a simulation tool built from scratch, fully supporting the simulation of MANETs.

Figure 1.1 depicts a coarse system overview. The development of the system was split in two theses. The core system, comprised of the *Simulation Engine* and the *algorithms*, was developed in [Molt04]. In this thesis, the *Mobility Engine*, the *Spatial Model* and the *Demonstrator Engine* were developed. These components allow the support of node movement, modeling of spatial constraints and user interactivity.

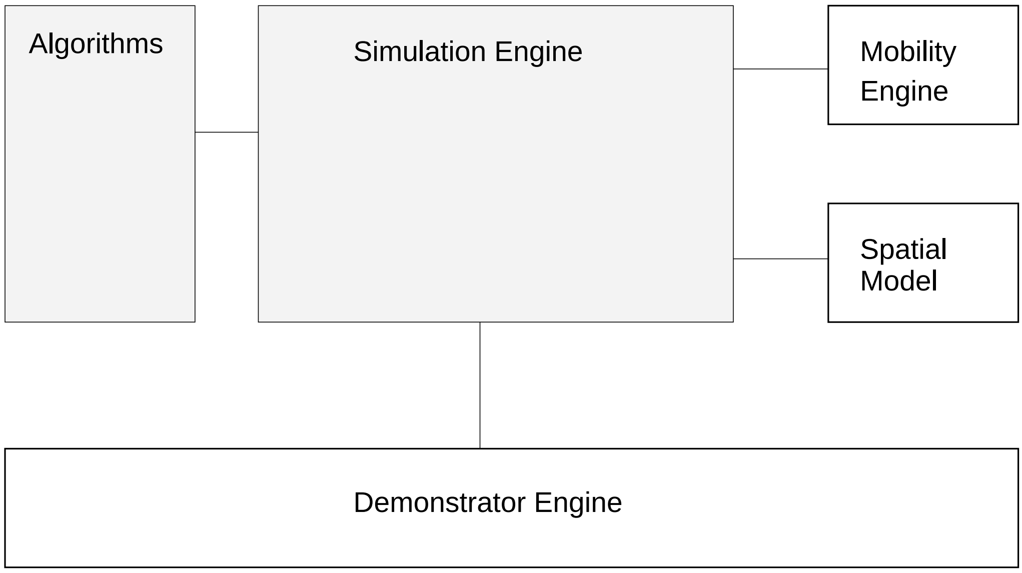

Figure 1.1: Coarse system overview of Cubus

1.1 Thesis Layout

In Chapter 2, we give some basics regarding modeling of mobility and radio propagation models, introducing the concepts we use troughout the thesis. Chapter 3 describes the architecture of the existing Cubus system and lists the requirements for this thesis. Chapter 4 discusses related work. We will examine different simulators, their architecture, and how they handle node mobility, animation, and interaction.

Chapter 5 and Chapter 6 describe the design of the mobility engine, demonstrator engine and the spatial model in detail. The environmental aspects and the components dealing with the mobility of nodes are discussed extensively in Chapter 5. The components handling user interactivity and analysis aspects are covered in Chapter 6.

Chapter 7 describes the current state of the implementation. The components implemented in this thesis have been evaluated using the scenario in [Molt04]. The results are discussed in Chapter 8. Chapter 9 concludes our thesis and gives a brief outlook on how to improve and extend some of our components.

2 Preliminaries

In this chapter, we briefly examine mobile ad hoc networks (MANETs) and their constraints. We discuss and classify mobility models and examine radio propagation and location models.

2.1 Characteristics of Mobile Ad Hoc Networks

In the past few years, wireless networks have gained great influence due to the decreasing costs of hardware equipment. An important class of wireless networks are *mobile ad hoc networks (MANETs)*. Mobile ad hoc networks consist of a set of mobile nodes, which form the network routing infrastructure in an ad hoc fashion. These networks are inherently different to static networks. According to [CoMa99], the following constraints are characteristic for mobile ad hoc networks:

1. The network has a dynamic topology, since nodes may move arbitrarily.
2. The bandwidth of mobile nodes has significantly lower capacity than their hardwired counterparts. Due to propagation path effects, the signals are unreliable.
3. A mobile ad hoc network has to deal with energy constraints, since mobile nodes rely on battery and therefore have only limited power and lifetime.

All these characteristics require new routing and data management algorithms, which are able to cope with the constraints of mobile ad hoc networks. Before these algorithms are deployed on networks, they have to be validated. However, testing the algorithms in the physical world can become very costly. In [MaBJ99], some of the problems of testing algorithms in a so- called "physical test bed" are pointed out: First, the infrastructure must be planned. However, due to the unreliability of signal propagation and other physical effects, like weather conditions and mobility of the objects, the results of a scenario run cannot be repeated exactly. Secondly, the whole scenario has to be set up which can become very costly, since all equipment has to be set up from scratch.

To overcome the drawbacks of testing in the physical worlds, network simulators are an alternative. Once a scenario is set up, it can be repeated multiple times.

2.2 Simulation of Mobile Ad Hoc Networks

In a network simulation, the complete scenario is implemented in software, comprising the network architecture as well as the algorithms which are under test. Simulation of mobile ad hoc networks has a series of advantages. The simulation runs are repeatable. It is much cheaper to set up a simulation than setting up a physical test bed. Moreover, the user can add analysis capabilities to visualize the obtained results.

When simulating mobile ad hoc networks, we have to consider aspects like the mobility of nodes and environmental aspects. These aspects differ from the simulation of static networks. First of all, the mobility of nodes has to be modeled. This is usually done by a function of time which computes the position of a node at a specific time according to its speed and previous position. In the next section, we examine how to model node movement.

Furthermore, the environmental aspects, like signal propagation, have to be modeled. Therefore, many radio propagation models have been developed during the last years which try to mimic signal propagation as realistically as possible. We briefly consider some of them in Section 2.4.1.

2.3 Mobility Models

When modeling mobile networks, one has to take the mobility of nodes into account. The movement should be close to reality in order to gain realistic results. A mobility model dictates how the nodes move within the network. During the years, many mobility models have been proposed.

The authors of [CaBD02] classify models into two different groups:

- Traces or deterministic models
- Synthetic models

Traces represent mobility patterns that are observed in the physical world. They are created by logging the movement behavior of real-life objects, like people, providing realistic information of a node's position. However, for new network environments, e.g. ad hoc networks, traces do not exist and it takes long observation periods to create traces.

To overcome this drawback of trace files, synthetic models were developed. They model the movement of a node without the need for traces. According to [Step02], we distinguish between *random models* and *hybrid models*. The classification is depicted in Figure 2.1.

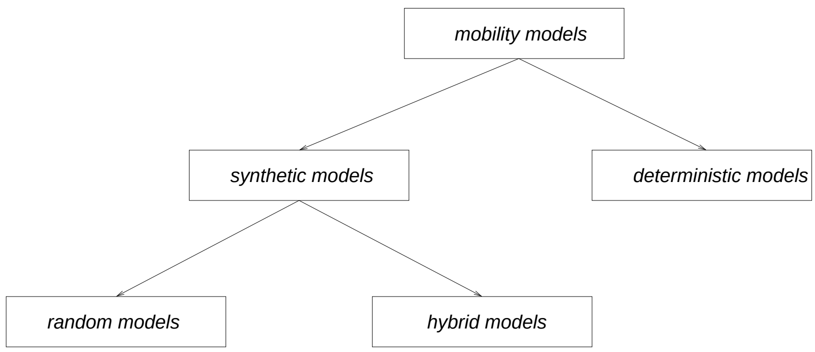

Figure 2.1: Classification of Mobility Models (adapted from [Step02])

Random models rely on random movement of nodes. Spatial constraints, like walls or streets, are not considered. The hybrid models try to model movements as realistically as possible by considering spatial constraints. They are a trade-off between the simple random models and the accurate trace files.

Node movement can be defined for each node independently (*entity mobility model*) or for a group of nodes (*group mobility model*). In the next subsections we take a closer look at two mobility models, that were implemented in Cubus. For a more comprehensive overview of mobility models see [Step02, CaBD02, Davi00].

2.3.1 Random Waypoint

The Random Waypoint Mobility Model [BMJH+98] is commonly used in network simulators, as it is very easy to implement.

At the beginning of the simulation, a node randomly chooses a destination in the simulation area and moves with a given speed (between V_{min} and V_{max}) towards it. When arriving at the destination, the node stays for a randomly chosen pause time (between P_{min} and P_{max}) and then the movement continues.

One major drawback of this model is that spatial constraints are not considered. In the physical world, people or cars don't move randomly. Another drawback pointed out by [YoLN03] is that the average node speed decreases consistently towards V_{min}. When V_{min} is chosen very small, the nodes get slower as simulation time proceeds, causing the system to become static. Hence the simulation results become unreliable. To avoid this problem, V_{min} should be chosen greater than 0.

2.3.2 Graph-Based Mobility Model

The Graph-Based Mobility Model [THBS+02] attempts to mimic behavior of people in the real world by using a graph to model the movement constraints.

Vertices represent points of interest, to which the user might want to move to. Egdes represent connections between these points. Similar to the Random Waypoint Model, the model has three parameters: The maximum and minimum speed a node can take, V_{min} and V_{max} respectively, and a pause time.

Each mobile node is initialized at a random vertex and moves towards another randomly chosen vertex. When arriving at the destination vertex, the node pauses for a selected period of time and then picks another vertex to move to.

As with the Random Waypoint Model, the average node speed decreases to V_{min}. Therefore, V_{min} should be chosen greater than 0.

2.3.3 Discussion of Mobility Models

When running a simulation with mobile nodes, there will be a trade-off when selecting a mobility model. If we do not need to consider spatial constrainst and we want our simulation to run fast, we use the Random Waypoint Model. If we want to obtain more realistic results, we use the Graph-Based Mobility Model.

2.4 Environmental Aspects

Besides modeling node mobility, modeling the environment is also an important aspect when simulating MANETs. In the following, we examine models for signal transmission and briefly discuss location models.

2.4.1 Radio Propagation

In the previous thesis [Molt04], the aspects of signal propagation are described. Like in wired networks, there are senders and receivers of signals. However, due to influences of the physical world, the signal strength at the receiver varies. If there is a vacuum between the sender and the receiver, three ranges can be distinguished, as depicted in Figure 2.2.

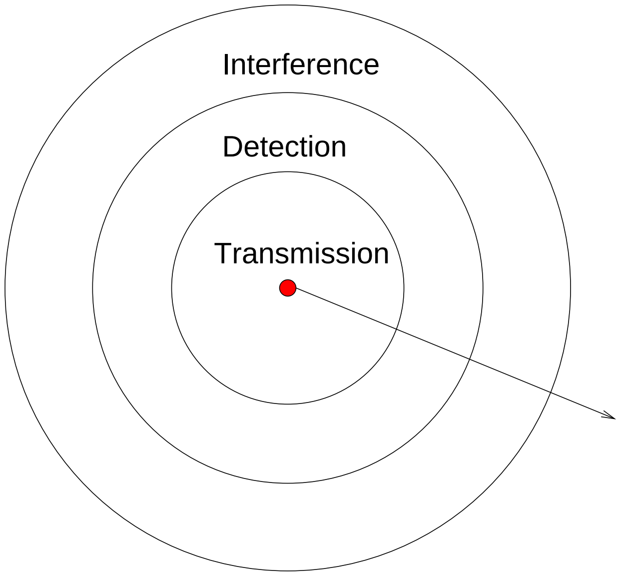

Figure 2.2: Ranges for transmission, detection and interference of signals (adapted from [Schi00])

If a receiving node is located inside the *transmission range*, it receives the signals with an error rate low enough to be able to communicate. Within the *detection range*, the error rate is too high to establish communication. In the *interference range*, the sender may interfere with other transmissions: A receiver will not be able to detect the signals, but the signals may disturb other signals.

Since most communication in wireless networks is done through air, signals travel through rain and dust particles, undergoing path loss. An extreme form of path loss is blocking or shadowing due to large obstacles, like walls. Furthermore, obstacles can reflect the signal.

All these effects have to be modelled in a simulation in order to gain realistic results. A radio propagation model mimics the signal transmission and calculates the signal strength at the receiver according to the model's underlying algorithm. In the following, we examine three simple propagation models which are implemented in the Network Simulator 2 (NS-2).

Free Space Model

The Free Space Model assumes a free line-of-sight between a sending and a receiving node. The transmission range is represented as a circle or sphere. The Free Space Model calculates a signal's power at the receiver by the *inverse square law* which concludes that the received power P_r is proportional to $1/d^2$ with d being the distance between sender and receiver.

Equation 2.1 gives the calculation according to [Frii46]. Signal strength at the receiver denoted by P_r is computed, taking into account the antenna gains of the sending and the receiving node denoted by G_t and G_r respectively , the transmission power of the sending node denoted by P_t, the system loss denoted by L, the distance d between the sending and receiving node, and the wavelength λ.

$$P_r(d) = \frac{P_t G_t G_r \lambda^2}{(4\pi)^2 d^2 L} \tag{2.1}$$

Two-Ray Ground Model

The Two-Ray Ground Model [Rapp96] considers not only the direct path between two nodes, but also an additional ground-reflection path. The received power P_r at distance d is computed by Equation 2.2, where h_t and h_r are the heights of the sending and receiving antennas. G_t and G_r are the antenna gains of the sending and receiving node and L is the system loss which is assumed to be set to 1.

$$P_r(d) = \frac{P_t G_t G_r h_t^2 h_r^2}{d^4 L} \tag{2.2}$$

When dealing with short distances, oscillations occur. Therefore, a cross-over distance, d_c, is computed by Equation 2.3.

$$d_c = \frac{(4\pi h_t h_r)}{\lambda} \tag{2.3}$$

If $d_c < d$, Equation 2.1 is used, otherwise Equation 2.2.

Shadowing Model

The Shadowing Model [Rapp96] takes multipath propagation effects into account. It comprises two steps. In the first step, the path loss is computed, i.e. the mean received power at distance d. This equation is shown below: β is the *path loss exponent*, $\overline{P_r(d)}$ denotes the mean received power at distance d, d_0 the cross-over distance, and $P_r(d_0)$ the strength of the signal at the receiving node.

$$\left[\frac{\overline{P_r(d)}}{P_r(d_0)}\right]_{dB} = -10\beta \log\left(\frac{d}{d_0}\right) \tag{2.4}$$

A second step reflects the variation of the received power at a specific distance.

2.4.2 Discussion of Radio Propagation Models

As with mobility models, there will be a trade-off when selecting a radio propagation model: If the primary requirement for the simulation is to run fast, a simple propagation model, such as the Free Space Model, is appropriate. In cases where the simulation results need to adhere as closely as possible to the physical world, we use a more complex radio propagation model that takes spacial constraints into account.

2.4.3 Location Models

A *location* is defined as an arbitrarily, uniquely defined area of the physical world. According to [Dros03], a *location model* defines the mapping between locations from the real world onto coordinates in a specific format. In [Domn01], two types of location models are distinguished which support different coordinate systems.

In *geometric location models*, locations are represented by n-tupels of geometric coordinates. Geometric location models are distinguished into *local geometric* and *global geometric* location models. A global geometric location model represents its locations according to a global reference system in the form of <latitude, longitude, altitude> triples, e.g. the WGS84 coordinate system [Euro04]. Local geometric models define their locations according to a local reference system.

Symbolic location models represent locations by symbolic identifiers. *Hybrid location models* combine both types of model. Every coordinate in a hybrid model is represented as a geometric coordinate as well as a symbolic identifier.

A location model contains the information to perform the transformation of a position, given in one format, into the format which the location model supports. Local geometric location models and symbolic location models may have a restricted scope. Only positions which are in the scope of the location model, can be transformed. Otherwise, an invalid position is returned. The mapping is defined within the location models. Global geometric location models define the mapping based on coordinate transformations. In [Open01] some example mappings are described.

To map geometric positions onto symbolic identifiers, the symbolic identifiers must have a corresponding geometric figure. In [DüRo03] a hybrid location model is described, in which each symbolic identifier is associated with a geometric figure. The symbolic identifiers are built hierarchically and are either associated with a geometric figure directly or inherit the geometric figure of their parents. To translate a symbolic location to a geometric figure, the associated geometry of the symbolic location is used.

2.5 Summary

In this chapter, we have discussed the notion of mobile ad hoc networks. These networks consist of a set of mobile nodes, which form the network routing infrastructure in an ad hoc fashion. The topology of these networks is very unstable due to the movement of nodes. Moreover, the transmission of signals is unreliable, since signals may undergo propagation effects. When simulating algorithms and protocols for these networks, we have to consider node mobility and spatial constraints in order to get realistic simulation results.

Network simulators model the movement of nodes by using mobility models. We distinguish two kinds of mobility models: Traces represent mobility patterns observed in the physical world. Synthetic mobility models define the movement of nodes without the need for traces.

Radio propagation and the path loss a signal may have to cope with, is modeled by radio propagation models. We have examined three propagation models implemented in NS-2.

Finally, we have briefly discussed location models which contain the information to perform the transformation of a position given in one format, into the format which the location model supports.

3 Cubus

In this chapter, we introduce the event-based simulator Cubus we developed. First, we briefly outline the core components, which were developed in a previous thesis [Molt04]. Secondly, we list the requirements that still need to be fulfilled. In the third section, we briefly discuss the components, we have to deal with in this thesis.

3.1 Existing Cubus System

Cubus is an event-driven simulator built to be highly modular. Each component provides a well-defined interface. Its main design focus relies on mobile, wireless networks but it also supports static networks. It support dynamic loading of components during runtime and hence has very small resource needs. Figure 3.1 depicts the architecture of Cubus. The *simulation engine* and *algorithms* are part of the previous thesis [Molt04] and are

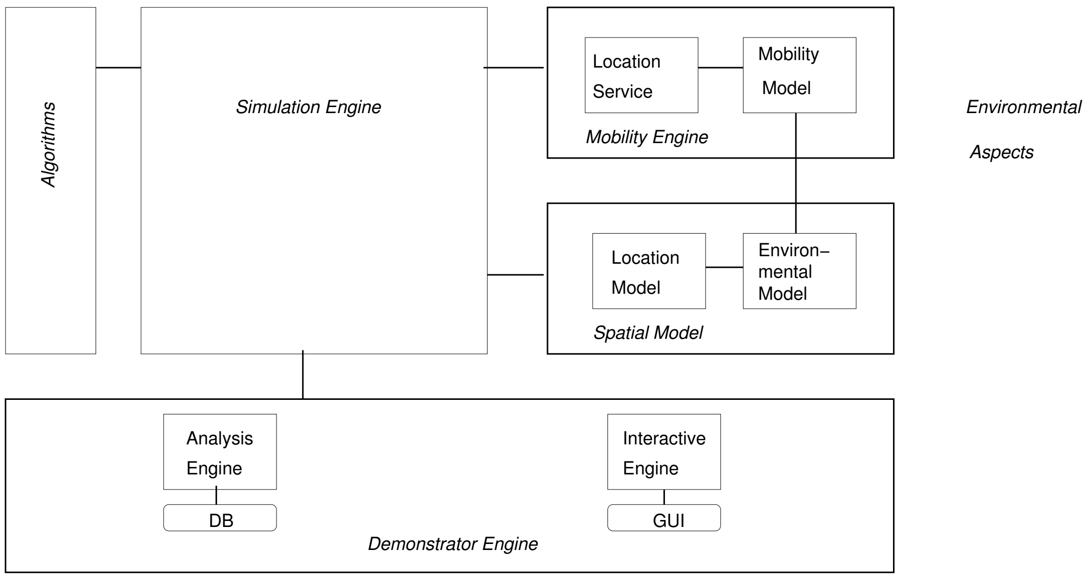

Figure 3.1: Cubus system overview (adapted from [Molt04])

discussed in depth there. This thesis deals with three components: the *mobility engine*, the *spatial model* and the *demonstrator engine*. Additionally, we discuss *positioning devices* and *propagation models* with respect to spatial constraints.

3.1.1 Simulation Engine

The simulation engine is the core of the Cubus system. It is comprised of the *core engine*, the *node runtime engine*, the *algorithm engine* and the *medium engine*. Figure 3.2 depicts a more detailed view of the simulation engine.

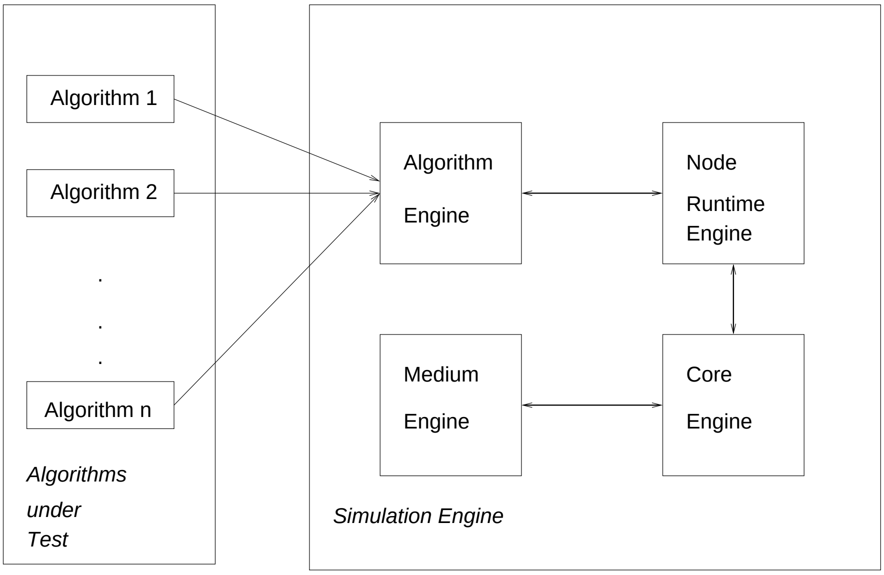

Figure 3.2: Detailed view of the simulation engine (adapted from [Molt04])

Core Engine

The core engine is responsible for event scheduling and updating simulation time. It manages two event queues for simulation and system events. System events are generated during a run of a simulation, but do not influence the continuation of the simulation. Simulation events are necessary for the ongoing simulation.

The core engine serves as a basis into which all other components of the simulator are plugged in.

Node Runtime Engine

Nodes are basic components in Cubus and represent all kinds of physical entities, e.g. mobile devices or workstations. Each node is associated with its own protocol stack and one or more network interfaces. Additionally, one or more positioning devices or any other kind of hardware devices can be plugged into a node.

Algorithm Engine

The algorithm engine provides the common interface for all algorithms. Cubus supports dynamic loading of algorithms, making it possible to exchange algorithms during runtime.

Medium Engine

The communication between nodes is carried out via a medium. The real-world counterparts are the air in wireless networks or a wire in wired networks. Our simulation media mimic the characteristics of their physical counterpart, e.g. error rate or bit rate. The medium engine provides a common interface for all media.

3.1.2 Algorithms

Algorithms are components which carry out a well-defined task, e.g. routing, data management.

By connecting algorithms, a protocol stack is built. Through these means the, ISO/OSI reference model can be constructed. Cubus is not restricted to this model. We support an arbitrary number of layers.

Each layer consists of an arbitrary number of algorithms. A packet traverses the stack upwards by calling the method `receive` of the upper-layer algorithm and removing its own header. Downwards the stack, the method `send` is called and the calling algorithm adds its header.

Each algorithm has properties we might want to access during runtime, e.g. queue size. The properties of an algorithm are described in a configuration file. This file is parsed at simulation startup and the information is stored in a repository.

To gain access to these parameters, the base class for algorithms defines generic *get* and *set* methods, which must be overwritten by the algorithm implementor.

3.2 Requirements

In the last section, we gave a brief overview of the existing Cubus system and the functionality it provides. In the following section, we discuss the requirements that still need to be fulfilled in order for our simulator to be tailored to mobile ad hoc networks.

Requirement 1: *Support for mobility*

When simulating mobile ad hoc networks, the movement of nodes has to be simulated. We need to provide mechanisms to easily plug arbitrary mobility models into the system. In NS-2 and GTNetS, nodes can only move individually. Since some scenarios, such as a group of people moving around in a museum, also require the simulation of group movement, Cubus needs to support group-based mobility models, as well as entity mobility models.

Requirement 2: *Modeling of spatial constraints*

In the physical world, mobile objects are bounded by spatial constraints, e.g. people usually do not walk through walls, but walk on sidewalks. Our simulator should support the modeling of such constraints.

Signals are the physical representation of data. When simulating wireless networks, one has to model signal propagation. Many simulators only support simple models, where the signal looses strength proportionally to the distance between sender and receiver (Free-Space model [Frii46] or Two-Ray-Ground Model [Rapp96]). In reality, a signal can be reflected or scattered from osbtacles [Schi00]. In order to get more realistic simulation results, we need to support the modeling of more complex propagation models which take obstacles into account. Therefore, we have to provide means to model propagation obstacles.

Requirement 3: *Support for Location Modeling*

Mobile nodes determine their current position by using positioning devices. Commonly used for outdoor positioning is the Global Positioning System GPS [GPS95] which uses global WGS84-coordinates [Euro04]. In order to simulate all kinds of positioning devices, we need to support location models of various kinds. The implementation of these models is responsible for the mapping between different coordinate systems. We should support local and global geometric coordinates as well as symbolic coordinates.

Requirement 4: *Support for User Interactivity and Animation Control*

Most simulators lack provisions for user interactivity in the simulation run. Our system needs to provide means to let the user interact with the system. The user should be able to control the animation speed and to change system parameters during runtime, as well as change algorithms or mobility models.

Requirement 5: *Provide Analysis Capabilities*

Popular simulators provide logging techniques for later retrieval of information. Like NS-2 and GTNetS, our simulator also needs to support fine-grained logging as well as to provide mechanisms to store logs for later retrieval. The storage location of the logged data should be chosen by the user. Cubus should support at least text files, as well as storage of logged data in relational databases.

3.3 Components of This Thesis

This section examines the components to be developed in this thesis in order to fulfil the requirements mentioned above. We briefly discuss which requirements are fulfilled by the components.

3.3.1 Mobility Engine

The mobility engine is comprised of the *location service* and the *mobility models* and deals with all aspects of mobility, such as node movement and the retrieval of a node's position. It fulfills requirement 1, the support of mobility.

Mobility Models

A mobility model defines the movement of its associated nodes. We provide a common interface for all mobility models. Like algorithms, mobility models have properties we might want to access during runtime of the simulation. Hence, the properties of a mobility model are defined in a separate configuration file and are stored in a repository.

Location Service

The location service is the interface responsible for all requests regarding the position of nodes. It supports position queries, where a component requests the position of a specific node. Moreover, it handles range queries and returns all nodes which are in a geographic range of a specified node. The geographic range is represented by an arbitrary geometric shape. The location service can be seen as a connector between the mobility models and the rest of the Cubus system.

3.3.2 Spatial Model

The spatial model deals with all aspects of spatial constraints, like obstacles. It is comprised of the *environmental model* and *location models*. Requirements 2 and 3 are fulfilled by this component.

Environmental Model

The environmental model includes the spatial constraints of the simulation. It is expandable, so that users can add more complex constraints to their simulation scenario. Besides mobility areas, on which nodes move, it contains propagation and GPS obstacles. In order to support any kinds of positioning devices, it may contain an arbitrary number of location models.

Location Models

A location model defines the transformation of positions given in one format, into positions of another format, e.g. symbolic coordinates. We distinguish between two kinds of location models. A geometric location model transforms geometric positions, e.g. into WGS84 coordinates [Euro04]. A symbolic location model transforms a position, given in geometric coordinates, into symbolic coordinates. In Cubus, both kinds of location models are supported.

3.3.3 Demonstrator Engine

The demonstrator engine adds interactivity and analysis capabilities to our simulator and fulfils requirements 4 and 5. It is comprised of the *analysis engine* and the *interactive engine*.

Analysis Engine

The analysis engine is responsible for the storage of the logged events. For the storage location of the logged data, we support not only text files, but also relational databases.

Interactive Engine

The interactive engine is the component which provides user interactivity and control of the simulation. By changing system parameters or exchanging algorithms or mobility models, the user can actively influence the simulation.

3.3.4 Further Components

Positioning devices and propagation models also take spatial constraints into account. Therefore, we mention them here separately.

Positioning Devices

Positioning devices compute the position of their associated node. Some devices, like GPS, don't compute the position exactly, but introduce additional errors. Moreover, positioning devices support only a specific coordinate system. Therefore, we add location models to our system which are able to map between different coordinate systems.

Propagation Models

A propagation model determines the propagation of transmission signals and the set of nodes in the transmission range of a sending node (or respectively, a network interface). We support simple propagation models, e.g. the Free-Space model [Frii46], as well as more realistic models.

4 Related Work

In this chapter, we give a short overview of the architecture of NS-2 [FaVa03], GTNetS [Rile03] and OMNeT++ [Varg01]. Next, we examine, whether these network simulators conform to the requirements we have listed in the previous chapter, like mobility of nodes and the support of spatial constraints. We also discuss whether they have animation or interaction support for the user and to which level they provide logging and analysis techniques.

4.1 Network Simulators

Network simulators have gained widespread acceptance for evaluating protocols and algorithms throughout the years. As all computations can be done in one place, only one computer is necessary. Furthermore, complex hardware is not needed. Hence, simulation is very cheap compared to testing protocols in a real deployment. Additionally, the user can run the same scenario more than once and can compare the results.

Network simulators have emerged from the field of wired networks, e.g. the Network Simulator 2 (NS-2), one of the first network simulators. With the advent of mobile and wireless networks, network simulation has focussed on the simulation of protocols and algorithms in this field. However, the simulation of these networks have further requirements, like the support of node mobility and the modeling of signal propagation.

There are two categories of network simulators. The first category is comprised of network simulators enhanced with new features to simulate mobile, wireless networks, e.g. NS-2 [FaVa03]. The second category is comprised of network simulators tailored for mobile, wireless networks, e.g. the Georgia Tech Network Simulator (GTNetS) [Rile03].

In the following, we briefly present an overview of three network simulators. A more detailed view on GTNetS and NS-2 can be found in [Molt04].

4.1.1 Network Simulator 2 (NS-2)

The Network Simulator 2 (NS-2) is an object-oriented discrete-event simulator for networking research based on the REAL simulator. It is comprised of a compiled class hierarchy of objects which is written in C++ due to efficiency reasons. Moreover, it contains an interpreted class hierarchy of objects, written in OTcl which is an object-oriented extension to the Tcl script language [RaTr99].

The simulation results are written to a trace file which can be traversed by the Network Animator Nam [EHHM+99]. Nam provides animation for the NS-2 simulator. Events are a crucial part of the simulator and consist of a globally unique ID, a timestamp and a pointer to the object which handles the event. The CMU Monarch project [Mona98] enhances NS-2 to support mobile, wireless networks. In Figure 4.1, the architecture of the extension is depicted. Each mobile node is an independent entity responsible for computing its own position. Moreover, an arbitrary number of network interfaces can be attached

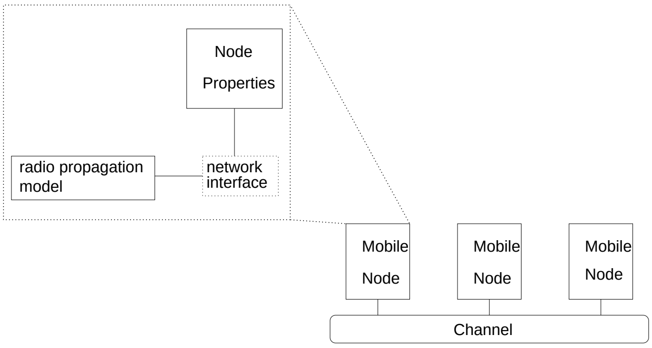

Figure 4.1: Monarch model (adapted from [Molt04])

to a node. As depicted in Figure 4.1, each network interface is attached to a channel. The channel carries packets between the mobile nodes and is characterized in terms of bandwidth and delay. Packets on different channels do not affect each other. Normally, all network interfaces of a specific kind are attached to a specific channel type. The packet of a sending node is copied to all other network interfaces on the channel. These network interfaces then each use a radio propagation model in order to determine whether or not the packet can be received.

Due to the two-split programming model, NS-2 has a high resource consumption, since the method calls of the OTcl objects have to be dispachted to the corresponding C++ class. As it has grown over the years, it also lacks modularity. All protocols to be tested, need to be included into the simulator at compile time. It is not possible to load algorithms or protocols at runtime. Another drawback is that building a new feature into NS-2 requires good knowledge of the internals.

4.1.2 The Georgia Tech Network Simulator (GTNetS)

The Georgia Tech Network Simulator (GTNetS) [Rile03] is developed by a research group from the Georgia Tech University. Its development started in 2002. It tries to overcome the deficiencies of NS-2, mainly its runtime overhead. Its main foci are scalability and distribution abilities, i.e. the simulation can be run in parallel. It is an event-based simulator which is completely written in C++. There is no configuration file, but users write their own `main` method when creating scenarios. As depicted in Figure 4.2, three major components are distinguished. Nodes are connected via network interfaces to links. Links represent the communication media via which nodes communicate.

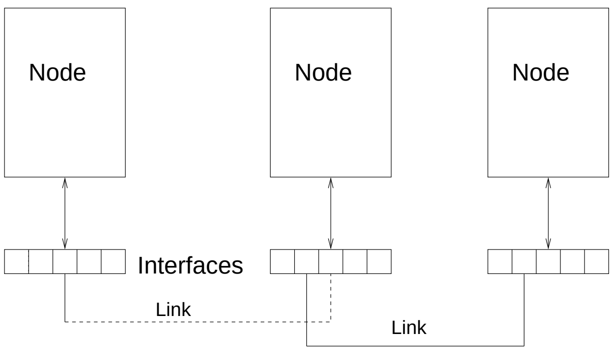

Figure 4.2: GTNetS architecture (adapted from [Molt04])

Each node is associated with a protocol stack. The protocols stack is restricted to the ISO/OSI model and cannot be extended by the user. The protocol designer has to derive the protocol from one of the protocol layer classes. In the current version, GTNetS supports various protocols at different layers, e.g. TCP, IP, IEEE 802.11. Since the development is still in progress, additional protocols are to be expected.

4.1.3 The Objective Modular Testbed in C++ (OMNeT++)

The Objective Modular Network Testbed in C++ (OMNeT++) [Varg01] is a modular discrete event simulator. It is not only suited for network simulation, but also for computing performance aspects of complex software. We mention OMNeT++ in this context, because it supports user interactivity.

A simulation scenario consists of hierarchically nested modules, which communicate with each other by passing messages. The modules at the lowest level are written by the user in C++. Unlike NS-2 and GTNetS, OMNeT++ does not only support discrete event processing, but the modules can also be realized as coroutines, which can be regarded similar to processes. Hence, OMNeT++ supports two kinds of simulation modes, the event-based and the process-oriented simulation [Molt04, Page91]

Figure 4.3 depicts an architectural overview of the Omnet system. *SIM* contains the core

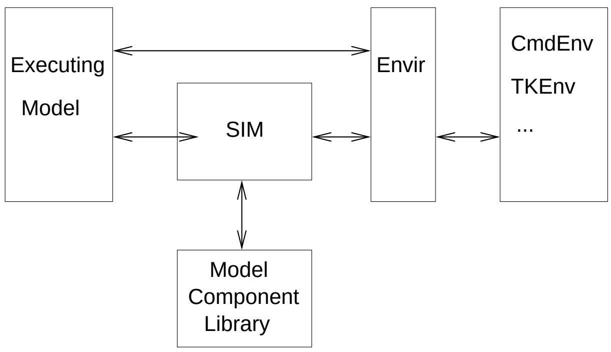

Figure 4.3: OMNeT++ architecture (adapted from [Varg01])

simulation classes. *Envir* provides a common interface for all user interfaces. At the current state, OMNeT++ supports two different user interfaces, a command-line user interface and a graphical user interface, written in Tcl/Tk. The *Model Component Library* contains all available simple module implementations and basic types, like channels, message types and network implementations. The *Execution Model* is the model set up for the simulation and is constructed from classes of the Model Component Library.

OMNeT++ is not focused toward specific applications. Support for the simulation of mobile ad hoc networks is obtained by enhancing the systems with module implementations and frameworks which are tailored to the characteristics of these networks. Many frameworks and extensions can be found on the World Wide Web. However, their documentation do not provide enough information for this thesis. Therefore, we examine only the Mobility Framework [LoWi04], which is developed by Telecommunication Networks Group at the Technische Universität Berlin and comes with extensive documentation.

4.2 Mobility Modeling

One characteristic of mobile wireless networks is the movement of nodes. When simulating these networks, we have to map the mobility of real-world nodes to the simulation. The movement of nodes is defined by mobility models. In the following, we examine how common network simulators model the mobility of nodes.

4.2.1 Mobility in NS-2

In the current version of NS-2, mobile nodes move on a flat topology. There are two mechanisms to define node movement.

In the first mechanism, the user can explicitly set the starting point and further destinations of a node in a separate trace file with the format: *$ns at $time $node setdest <x2> <y2> <speed>*. These tracefiles can be generated by external tools, e.g. the Canu Mobi Sim environment [SHBT+03] or the CMU Monarch scenario generator [Mona98]. These tools generate trace files in the format of NS-2 by applying mobility models.

The second method employs random movement on nodes. The mobile node starts at a randomly defined start position. When updating a node's position, a destination point and speed are randomly chosen.

Trace files can be generated from more complex mobility models, e.g. by the use of the tools mentioned above. The drawback of this approach is that the user has to use external tools, in order to generate trace files of more complex mobility models. These tools might be complex to use and be very time-consuming. For example, the CMU Monarch scenario generator automatically computes optimal routing and hence the computation of a trace file containing 500 nodes takes up to two hours on Sun workstation with 1,2 GHz.

Another drawback is that trace files are only suitable when there is no support for user interactivity. When supporting user interactivity, the user might change parameters, e.g. the speed of a node. Since all entries in a trace file are precomputed , they would then become obsolete.

4.2.2 Mobility in GTNetS

GTNetS supports wired and wireless networks. Nodes are assumed to be static. The user has to declare mobile nodes by associating a mobility model to them. Mobility models are defined in C++ classes.

In GTNetS, each mobile node is associated with its own mobility model instance. The mobility model determines the current position and velocity of the node as a function of time. The computation is not done continuously, but only when requested by other components. Hence, the overhead for computationally intensive mobility models is comparatively low. Nodes move in a two-dimensional topology which is represented by a rectangle or a grid. Three-dimensional topologies are not yet supported.

All mobility models are derived from a common base class which provides the methods to compute the position of a node and to retrieve parameters of the model, e.g the *maximum speed* at which a node can move. The current version of GTNetS only supports simple mobility models, which support random movement of nodes (*Random Walk Model* [HoAk95] and *Random Waypoint Model* [BMJH+98]). Since it does not support more complex topologies, it is difficult to integrate more complex mobility models. The user has to define the whole topology and to integrate this into the system.

4.2.3 Mobility in OMNeT++

Like NS-2, OMNeT++ is a simulator which has emerged out of the field of static networks. The Mobility Framework [LoWi04] is intended to support wireless and mobile simulations within OMNeT++. The general architecture of the Mobility Framework is depicted in Figure 4.4.

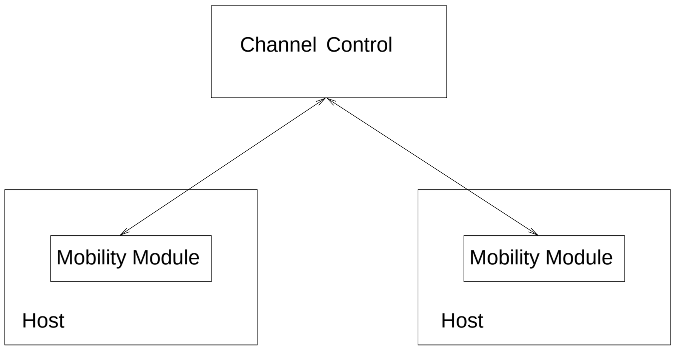

Figure 4.4: Mobility Architecture of OMNeT++ (adapted from [LoWi04])

Hosts represent any kind of mobile node. The global channel control module is responsible for establishing communication channels between hosts that are in communication distance and tearing down these connections when hosts move out of range. Each host module has an individual mobility module which handles the movement of the host and communicates the location changes of the host to the channel contol.

The framework has a common base class for all mobility models from which newly designed mobility models are derived. This base class provides methods for the placement of nodes in the simulation area and the movement of nodes. In the current version, only the *Random Waypoint Model* is provided.

4.3 Environmental Aspects

In the following section we examine how the environment of a simulation is modeled in NS-2, GTNetS and OMNeT++. We discuss whether there is any support for obstacles which constrain the movement of nodes or influence a signal's transmission. Moreover, we discuss whether positioning devices and location models are supported.

4.3.1 Environmental Aspects in NS-2

The simulation area is defined only as part of the CMU Monarch extension [Mona98] for mobile networks. It is defined in the configuration file for the generation of mobility traces. NS-2 only supports two-dimensional areas, e.g. rectangles or grids. The movement of nodes is defined in trace files. Hence, if a user wants to simulate nodes by a more complex model, that takes obstacles into account, he has to use an external tool, like CanuMobiSim [SHBT$^+$03], to generate the required trace files.

NS-2 does not provide any capability to model obstacles or any other kind of spatial constraints. Additionally, positioning devices and location models are not supported in the current NS-2 version. The position of a node is stored directly inside the node.

4.3.2 Environmental Aspects in GTNetS

Like NS-2, GTNetS does provide very little support for spatial constraints. Rectangles and grids are supported as simulation areas where nodes are placed. Only two mobility models are implemented, which do not take the environment into account (*Random-Waypoint Model* [BMJH+98] and *Random Walk Model* [HoAk95]). Also, there is no support for any kind of obstacles provided.

Like NS-2, GTNetS does not provide any mechanism to support positioning devices. The positions are stored in each node. A new position is computed when requesting the mobility model of the node. Location models are also not supported in GTNetS.

4.3.3 Environmental Aspects in OMNeT++

In the current version of the Mobility Framework, there is no support for spatial constraints. The nodes move in a rectangular area. Since the framework is still in development, this kind of support is expected in the near future.

Like NS-2 and GTNetS, the current version of the Mobility Framework does not support the modeling of spatial constraints. Positioning devices and location models are also not supported.

4.4 Radio Propagation Models in Simulators

4.4.1 Radio Propagation Models in NS-2

In NS-2, three different radio propagation models are realized. The *Free Space Model* [Frii46] considers a single line-of-sight between two nodes. The *Two-Ray Ground Model* [Rapp96] takes not only the direct path between two nodes into account, but an additional ground-reflection path. The *Shadowing Model* [Rapp96] takes multipath propagation effects into account. The transmission range and other properties of the network interface, eg. antenna gain or antenna height, can be defined in the scenario configuration file. However, NS-2 lacks propagation models, which take obstacles into account. These models, along with the obstacles, have to be modelled by the user

4.4.2 Radio Propagation Models in GTNetS

Radio propagation models are plugged into the simulation the same way as mobility models. They have a common base class from which newly designed models are derived.

In the current version, only the *Free Space Model* and the *Two-Ray Ground Model* are implemented. Like NS-2, it lacks the capability of adding more complex models, as there is no support provided for environmental aspects. Moreover, when adding more complex models, the whole system has to be recompiled.

4.4.3 Radio Propagation in OMNeT++

Unlike NS-2 and GTNetS, a radio propagation model in OMNeT++ is not represented by a class. It is realized in the submodule for the physical channel. The propagation of a signal is calculated with parameters, like the *transmitter power*, the *carrier frequency* and the *path loss* in order to determine the attenuation of the signal. However, the modeling of more realistic propagation models is constrained since propagation obstacles are not supported in the current version.

4.5 Animation and Interactivity in Simulators

In the following, we examine whether our simulators support animation and user interactivity. We examine, if there are any mechanisms provided for the user to control the animation speed.

In discrete event simulation, we distinguish between two notions of time [Ball96]. The *wall clock time (WCT)* represents the physical time between the start and the end of the simulation. The *simulation time* is defined by the timestamps of the events scheduled next. It may vary in progress and can be faster or slower than wall clock time. If it is equal to the wall clock time during the whole simulation, the simulation is run in real-time. In most cases, simulation time does not correspond to wall clock time, since the system may "jump in time". If there is no event scheduled for the next three minutes in simulation time, the system will move the system clock three minutes forward in one go. This means that the simulation time will often run faster than the wall clock time.

Most simulators provide a mechanism to animate components of the system, e.g. packet flow or the changes of system state. However, one has to cope with the animation speeding up or down because of uneven time jumps. Jumps may be separated by five seconds simulation time, but it could also take four minutes between two events. Hence, the animation often speeds up or down. It is desirable to have a smooth animation of the simulation run.

4.5.1 Animation and Interactivity in NS-2

NS-2 itself provides no animation capability, but relies on Nam, a Tcl/TK based animation tool for viewing network simulation traces and real world packet trace data. It interprets Nam trace files. These files contain static network layout and time-indexed dynamic events, e.g. packet arrival or packet drop. The traces can be generated from a NS-2 simulation run, but can also be generated by processing data from a real network.

Nam animates the trace file offline and in replay-mode. First, the network topology is built. Afterwards, packet animation is started. Trace events indicate when packets enter and leave channels and queues. To run the animation more smoothly, Nam introduces so-called "dummy-events".

Users can control the animation playback rate to focus on parts of the simulation, they are interested in. They can start and stop the playback, as well as skip parts of the run.

A main drawback of Nam is that animation in NS-2 is only possible after a simulation run. Animation of packet flow during a simulation run is not supported. Therefore, interaction with a running simulation is not possible. Whenever there is a change to parameters of a scenario, the Nam traces have to be regenerated by running a new simulation.

Animation and Interactivity in GTNetS

GTNetS supports a rudimentary graphical user interface written in Qt [Trol04]. The user interface acts as an event-handler and subscribes itself continuously to update events. Each time the user interface receives an update event, it displays the current positions of nodes and the current status of packets in the network and schedules a new update event. The interval between update events can be determined by the user. If the user chooses a small interval, the user interface will run very slowly. If the interval between two update events is large, many state changes will have occured in the system between updates, but the user interface will run faster.

The approach of GTNetS fits our needs best. The animation is done during a simulation run and the user can determine the animation speed. The only drawback is the lack of interaction in GTNetS.

4.5.2 Animation and Interactivity in OMNeT++

OMNeT++ provides TKEnv, a graphical user interface written in Tcl/Tk. TKEnv supports interactive execution of the simulation for debugging and teaching aims. The user is able to view simulation results during its execution in *inspector windows*. The animation speed can be controlled by the user, since TKEnv offers four animation modes. The *step-mode* executes the simulation event-by-event. In the *run-mode* the simulation runs with all tracing aids on, e.g. message animation. In the *fast-mode* and *express-mode*, the animation is turned off. In the fast mode, the user can interactively change parameters of the system about every 10 events. The express-mode is the same as running the simulation from a command-line user interface.

In order to update the state of the graphical user interface, the simulator synchronizes itself with TKEnv. In the step mode and run mode all inspector windows and other parameters are updated after each event. If the user opens a new window, it will be filled with the current values at the next event. In the fast-mode, TKEnv is updated every 10 events, in express mode the user can interact with the system only rarely (default are 1000 events).

In the following subsection, we briefly discuss the HiSAP system since it provides an interesting solution to the time jumps in discrete event simulators.

4.5.3 Overview of the HiSAP system

HiSAP (Highly interactive simulation of Algorithms and Protocols) [Brod00] is a framework to build programs which simulate and animate algorithms and protocols in the field of distributed systems. It is aimed at teaching purposes and offers animation control by the user. The system was developed in Java and consists of a *simulation component*, *visualization-and animation component*, *generators* and *user programs*. The most important part is the simulation component, which controls the flow of the simulation. The user is able to control the animation speed, and to pause and resume a simulation by using the animation and visualization component.

The simulation component is realized as a discrete-event simulator, where events are stored in an event list. There are two notions of time, the *virtual time* is the time of the next pending event. The *physical time* resembles the current wall clock time. The simulation component is realized as a thread which sets its future virtual time to the timestamp of the next pending event. The interval between two events resembles real-time. This is realized by pausing the thread for an amount of physical time. The physical time is computed by the current virtual time, the time of the next pending events and a speedup factor chosen by the user. For example, when there are three minutes between two events and the speedup factor is two, the thread will sleep for one and a half minutes. No event processing is done until the thread resumes.

As the system also needs to support events created by the user, the system has to resume the thread each time a user-defined event is scheduled. The already elapsed physical time has to be mapped to the virtual time, and the future virtual time must be recomputed. Consider an example adapted from [Brod00]:

"The current virtual time is *7.30* and the timestamp of the next pending event is set to *7.35*. As the speedup factor is two and the physical time is *3.45*, the thread is sent to sleep for two and a half minutes. After half a minute, the user creates an event which timestamp is set to *7.32*. The thread is resumed and computes the current virtual time to *7.31*. The future virtual time is set to *7.31*, since the event is the next pending event and the thread resumes for half a minute till *3.46*. At *3.46* physical time, the event will be handled by the simulation component. "

With the notion of a thread, the user is able to control the animation speed in HiSAP. This approach could be integrated in the core engine of Cubus, but it would lead to a slower simulator since the time interval between two events resembles wall clock time. Since we want our simulator to run simulations as fast as possible, we do not consider this approach any further.

4.6 Logging and Analysis Capabilities

When running a simulation, the user is interested in various data, like the number of packets sent in the network. Network simulators differ in the level of logging and the components which can be logged during a run.

Normally, the logged data is stored in text files and is afterwards processed by plotting tools, e.g. Gnuplot, or by applying further statistical classes, e.g. histograms on the data. In the next subsection we examine which logging and analysis capabilities NS-2, GTNetS, and OMNeT++ provide.

4.6.1 Logging and Analysis Capabilities in NS-2

NS-2 supports two different kinds of logging. *Traces* record each individual packet when it arrives, departs, or is dropped at a node, whereas *monitors* record counts of various interesting quantities, such as packet and byte arrival . The traces are stored either in a textfile or are send to the standard output. All traces contain the simulation time at which the events occured.

4.6.2 Logging and Analysis Capabilities in GTNetS

GTNetS support very fine-grained logging capabilities. Logged events are stored in a text file, where they can be further processed. Packet tracing can be enabled or disabled either at individual nodes or protocol objects, or globally at the protocol layer. The designer of a protocol defines the items to be logged. All traces contain the simulation time and the identifier of the node, processing the packet.

Like NS-2, GTNetS does not support the storage of logs in a relational database. A variety of statistic classes are supported, e.g. histogram. These classes can be used for the analysis of a simulation run.

4.6.3 Logging and Analysis Capabilities in OMNeT++

In OMNeT++, each object of type *cOutVector* can write data out to a common text file, where the data can be further processed. Each module which the user wants to be logged has therefore be derived from the cOutVector class. The storage of logged data in a relational database is not supported recently.

OMNeT++ provides a series of statistic classes which can calculate for example mean and standard deviation, and display histograms.

4.7 Discussion

In the preceding sections, we have compared the three network simulators NS-2, GTNetS and OMNeT++ regarding the requirements we have outlined in the previous chapter. Table 4.2 summarizes our results.

NS-2 supports all kind of mobility models. However, these models are not directly integrated into NS-2. The user has to generate trace files out of the models through the use of external tools. Trace files are suited for simulators which do not provide any kind of user interactivity. When supporting user interactivity, we have to expect changes in parameters, like the speed of a node. The entries in trace files would then become obsolete. Therefore, we consider the solutions of GTNetS and OMNeT++ to suid our needs best. In both cases, all mobility models are directly integrated into the simulator. They both provide a common base class for mobility models and hence offer a common interface for the designer of a mobility model. However, in the current versions of GTNetS and OMNeT++, no mobility model is realized which takes spatial constraints into account, or in which nodes move together as a group.

All three simulators lack the support for any spatial constraints, like modeling of obstacles which influence the movement of nodes or the transmission of a signal. Also, no positioning devices and location models are supported. Moreover, due to the lack of support for propagation obstacles, there is no support for more complex radio propagation models.

Animation of the simulation run is supported in all of the simulators we have examined. However, NS-2 supports only the replay of a simulation by use of the Nam tool. Both, GTNets and OMNeT++, support the real time animation of a simulation run. To ensure, that the user interface has the most current state of the simulation, both simulators use update events for which the user interface subscribes. When an update event is handled by the user interface, the simulation pauses until the newest state of the simulation is displayed. The user can change the interval between update events. This approach seems to fit our requirements well.

All three network simulators provide logging capabilities. Logged data is stored in text files. There is no support of other storage location, like relational databases. In all three network simulators basic analysis capabilities are supported, e.g. histogramms.

	NS-2	GTNetS	OMNeT++
support of arbitrary mobility models	✔	(✔)	(✔)
modeling of spatial constraints	✘	✘	✘
support of positioning devices	✘	✘	✘
support of location models	✘	✘	✘
support of more complex radio prop. models	✘	✘	✘
support of animation control	✔	✔	✔
support of user interactivity	✘	✘	✔
providing analysis capabilities	✔	✔	✔

Table 4.2: Overview

4.8 Summary

In this chapter, we have given a short overview of some commonly used simulators and their architecture. All of them are discrete event-based simulators. The Network Simulator 2 (NS-2) is the oldest and most established network simulator. It uses a two-split programming model, since the class hierarchy is built in C++, whereas scenarios are defined in OTcl by the user . This split leads to a high resource consumption. Additionally, NS-2 has a high learning curve for new users. The CMU Monarch project has enhanced NS-2 to support mobile wireless network simulations.

The Georgia Tech Network Simulator (GTNetS) is being developed since 2002 and tries to overcome the deficiencies of NS-2. Furthermore, it main focus is on wireless mobile networks. It is entirely implemented in C++ and therefore has a lower resource consumption than NS-2. Since it is still under development, there are not as many protocols and algorithms implemented as for NS-2.

OMNeT++ is a simulator which is not primarily intended for the simulation of networks. Besides discrete-event simulation, it supports a process-oriented approach. We have taken a closer look at the support of interactivity in OMNeT++.

We have shown how mobility is modelled in NS-2, GTNetS, and OMNeT++. In GTNetS and OMNeT++, each node is associated with its own mobility model. In the current version, only random models are provided which do not take spatial constraints into account. NS-2 follows another approach. The movement of nodes is defined in tracefiles. The traces can be generated by external tools. Hereby, more complex mobility models can be defined.

We have also examined the problems of animation and interaction in discrete event-based simulators. As the internal clock of the simulator does not grow linearly, but jumps in time, the control of animation speed is not trivial. We have discussed the HiSAP approach. The advantage of HiSAP is that the interval between two events resembles physical time. In GTNetS, the graphical user interface is updated after x events. The user can control the animation speed by choosing the interval between the updates.

5 Design of the Mobility Engine and the Spatial Model

The previous chapter concludes that in common network simulators there is only little support for more complex environmental aspects, like propagation obstacles. In this chapter, we discuss the design of the *mobility engine* and the *spatial model*. The mobility engine deals with all aspects concerning the mobility of nodes. The spatial model contains all spatial constraints of the simulation area.

5.1 Mobility Engine

In the previous chapter, we pointed out the importance of modeling node mobility. The mobility engine is the component in our simulator which deals with all aspects of mobility. It is comprised of the *mobility models* and the *location service*. Moreover, as *positioning devices* have to cope with the mobility of their associated node, they conceptually belong to this engine.

Mobility models define the movement of the nodes in the simulation.The location service is an interface for the components which are interested in the position of nodes. Positioning devices are requested by algorithms in order to get the current position of their associated node. In the following, these components are discussed in detail.

5.1.1 Mobility Model

A mobility model defines the movement of its associated nodes. It is able to compute a node's position at every point of time in the simulation. The computation occurs only on demand, since there is no thread which continuously computes the position during the simulation, making these models more efficient.

Since we want the user to be able to change parameters of the model, we adopt the approach of GTNetS to explicitely build mobility model in our simulator. The trace files based approach of NS-2 is only suitable when there is no interactivity. When the user changes the speed or position of a node during the running simulation, all entries in the trace file become obsolete. Nevertheless, trace files can be modeled by a mobility model instance in our simulator.

An arbitrary number of nodes can be assigned to an instantiated mobility model. This leads to a more efficient simulator and less memory usage, since several nodes share a common model. Additionally, we are able to support group-based mobility models by simulating group behavior.

By default , each node is assigned to a *static* mobility model. This model always returns the same position for the associated node. In the scenario file, the user can specify alternative mobility models for a node. Hence, the user can simulate static and mobile nodes in the same simulation run.

Like algorithms, mobility models have properties. We distinguish between properties of the model and properties of the associated nodes. The *maximum speed* is an example for the property of a mobility model. Properties of an associated node are e.g. *speed* and *position*.

As C++ lacks the capability of reflection, we have to publish the internal structure of the model to the Cubus system in another way: The properties are described in configuration files, which are parsed during start of the simulation. Afterwards, they are stored in a *repository*. To gain access to the properties of a specific instance of a mobility model, we have introduced generic *get* and *set* methods which have to be overridden by the implementor of a mobility model. These methods return or set the value of a given parameter respectively. For example, the call *get("maxSpeed")* returns the current allowed maximum speed a node can have.

Since all mobility models are derived from a common base class, we provide a common interface for all mobility models. This allows the easy exchange of mobility models during runtime. The base class defines methods to add and remove nodes from a mobility model which have to be overwritten for each model individually.

In the physical world, nodes move in an environment, e.g. cars are driving on a street. In network simulation, simulation areas model the environment of the nodes. In NS-2 and GTNetS, the simulation area is modelled by a rectangle or a grid. However, many mobility models require complex areas, e.g. the Graph-Based Mobility Model requires a graph. We introduce the concept of mobility areas.

Mobility Area

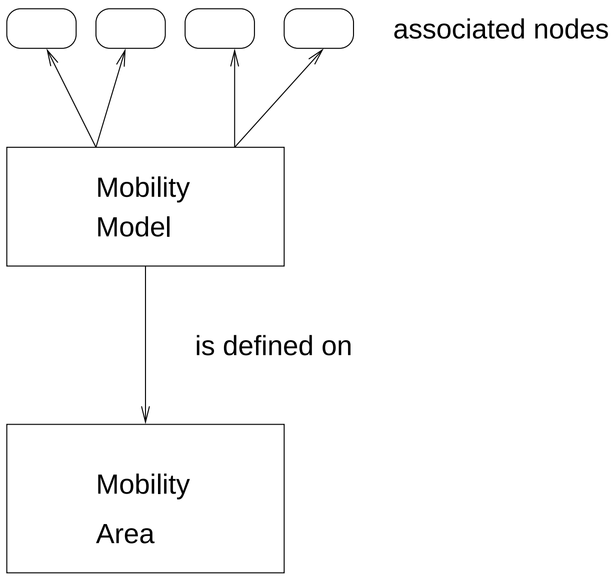

Figure 5.1: Mobility model with associated mobility area

Figure 5.1 depicts the dependency between the mobility model and a mobility area.

Therefor, a mobility area models the region, in which nodes can move. This region can be a simple two-dimensional rectangle or a three-dimensional cube respectively, or can be more complex, e.g. a graph. The mobility area is embedded in the environmental model and therefore is a part of the spatial model.

Each mobility model only accepts a certain type of mobility area, e.g. the Graph-Based Mobility Model is defined for a graph as underlying area. Some mobility models are only defined for 3D-areas, e.g. the 3D-Movement Model [TaBa99]. These models are associated with an area that is defined in 3D. With the notion of mobility areas, we are able to support complex mobility models, which take obstacles into account, e.g. the Obstacle Mobility Model [JBRAS05] by adding more complex areas, e.g. Voronoi Graphs.

When removing nodes from their associated mobility model and adding them to another mobility model, we have to take care of the placement of the nodes, since the current position of the node may not be defined for the area of the newly assigned model. This placement has to be defined by the mobility model designer. For example when removing a node from the Random Waypoint model and adding it to the Graph-based model, the point of interest on which the nodes is placed on, can be determined by calculating which vertex has the lowest distance to the node's current position.

During a simulation run, more than one mobility model can be instantiated. We can even instantiate the same mobility model on different areas, e.g. pedestrians and cars that are both simulated by a graph-based model, but move on different graphs.

5.1.2 Location Service

The location service can be seen as the interface for all components which are interested in the positions of nodes. It introduces some kind of abstraction for the components of the system, since they do not need to know the association between mobility models and nodes.

The location service supports *position*, *range*, and *distance* queries.

Position queries return the position of a specified node at the current simulation time. The location service consults the mobility model of the specified node to compute the position. Position queries are posed by positioning devices in order to get the current position of the node. Moreover, the core engine and the interactive engine are interested in the current position for displaying the node.

Range queries return all nodes in a specific geographic range of a sending node. A geographic range can be modeled for example, as a circle or sphere or by more complex polygonal shapes.

Distance queries return the euclidian distance of two specified nodes according to the following equation. Note, that when supporting only two dimensions, the z coordinate, representing the height, will be constant.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{5.1}$$

Propagation models use distance queries, in order to determine the strength of a signal at the receiving node.

A simple implementation of the location service stores all nodes of the system in a linear list. When a range query occurs, the list has to be traversed. This can become computationally intensive. We discuss data structures and optimizations the location service might use in order to overcome this problem. We first discuss the quadtree [Same84] as a possible data structure to make these queries more efficient.

Optimization issues

A quadtree (octtree) is a hierachical data structure. Each node in the tree has either four or eight children, respectively, or no children. Nodes without children are called *leaves*, whereas nodes with children are called *inner nodes*. Leaves have an in-degree of one and out-degree of zero nodes, inner nodes have an in-degree of one node and an out-degree of four or eight nodes respectively. The data is stored at the leaves, inner nodes only contain references to their children. The root node represents the whole simulation area. It is decomposed into four equal parts, each of them stored in child node. The areas saved in the leaves are called *service areas*. This hierarchical decomposition can be repeated several times until the resolution of the area is fine-grained enough. Figure 5.2 depicts an example decomposition and the according quadtree.

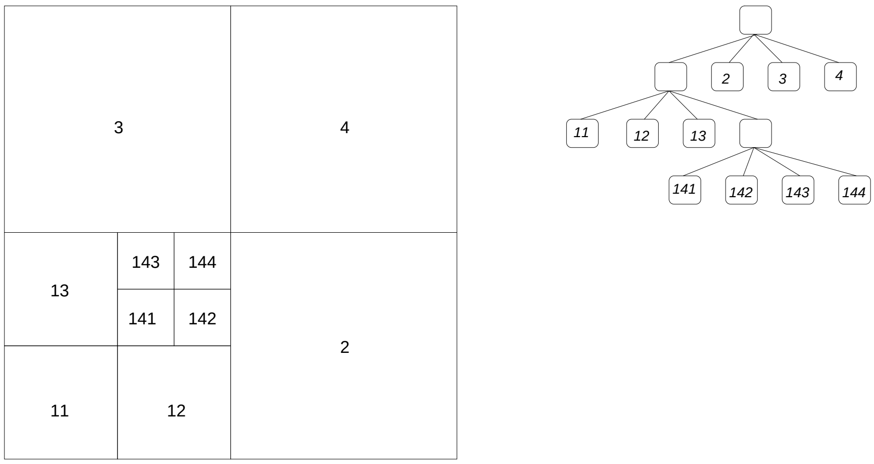

Figure 5.2: Decomposed simulation area and corresponding quadtree

A position query will be forwarded to the leaf whose service area contains the given position. The given position is determined by consulting a hash map which maps the node's identifier to its position. Assuming a balanced tree, this operation takes $\mathcal{O}(log_4(n))$ with n being the total amount of nodes in the quadtree. A range query request is forwarded to all leaves whose service areas overlap with the given range. All nodes within these leafs are returned. The query first retrieves the node's position by consulting the hash map. Afterwards, all nodes which are in the geographic range of this position are returned. Like position queries, the complexity of this operation is $\mathcal{O}(log_4(n))$.

Until now, we have assumed that the nodes in the simulation are immobile. However, when simulating mobile nodes, we have to cope with mobility issues. Nodes might move out of service areas and to another quadtree leaf. As the position of the nodes is not computed continuously, we have to update the quadtree when position or range requests occur. Especially range requests are interesting to discuss in detail, since the quadtree structure might loose its performance, when dealing with mobility.

There are several options to deal with this problem: The first one is to recompute the position of all nodes, when a range query occurs. This requires an effort of $\mathcal{O}(n)$. The quadtree would not be more efficient than storing all nodes in a linear list.

Another possibility is to use local updates. For each node, the location service stores the point of time at which the position of the node has been updated and the maximum speed the node might have. When receiving a range query, the location service computes only the nodes which might have changed their corresponding quadtree node. This does not lead to a more efficient data structure, as we still need to traverse all nodes.

Besides the quadtree structure, there are other possibilities for retrieving the position of a node as fast as possible. One is the precomputation of the positions of all nodes in the simulation. At startup the location service computes the position for each node at predefined time steps by using the associated mobility model. If a request is made for a position of a node, the location service either has a position entry for the given time or approximates the current position of the node. However, with this approach we face a problem when we support interactivity: If the user changes the speed or the mobility model of a node, all precomputed entries behind this point of time become obsolete and have to be recomputed. This can become very costly when there is a lot of interactivity.

In the last section, we have outlined a few remarks on how to store mobile nodes efficiently in network simulators. However, a more detailed in-depth look is beyond the scope of this thesis.

5.1.3 Positioning Devices

In the physical world, mobile devices are often equipped with positioning devices. These devices determine the current position of their associated node. It is desirable to model such devices in a simulator.

We distinguish between *geometric* and *symbolic* positioning devices. Geometric positioning devices return the position of a node as geometric coordinates. A well-known geometric positioning system is the Global Positioning System GPS [GPS95]. Symbolic positioning devices, like the Active-Badge System [WHFG92], return the position of a node in form of a symbolic identifier, e.g. a room number.

In Cubus, an arbitrary number of positioning devices can be attached to a node. Algorithms can query specific devices for the position of their associated node. The devices have a reference to the location service. Hence, they get the current position of the node without further knowledge whether the node is mobile and which mobility model is associated with the node.

Cubus provides a common interface for all positioning devices. Since our positioning devices should be as realistically as possible and should take spatial constraints into account, they have a reference to the component that stores propagation obstacles. A GPS device may add an additional error to the current location of the node. It computes whether the position returned from the location service is inside or outside a building. Figure 5.3 depicts the workflow when a GPS device is requested for the position of a node. If the node

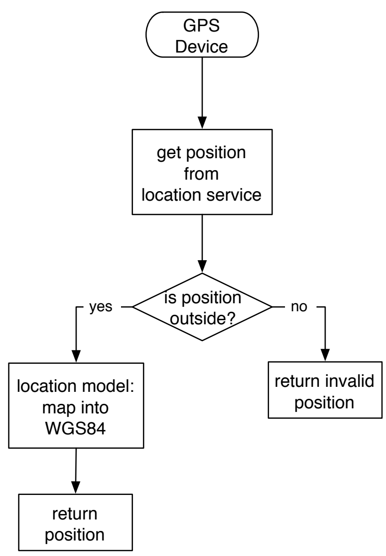

Figure 5.3: Position query of a GPS device

is located inside a building, the GPS device returns an invalid position, because it is not possible to receive GPS signals inside buildings. If the position is outside a building, the location model maps the position to WGS84-format. After that, the transfomed position is returned.

Symbolic positioning devices return the position of a node as a symbolic identifier, e.g a room number. Since positions in Cubus are internally stored in a geometric format, the symbolic device has to request a location model in order to transform the position. Figure 5.4 depicts the workflow when a symbolic positioning device, e.g. an infrared device, is requested by an algorithm for the position of a node. First, the infrared device queries the

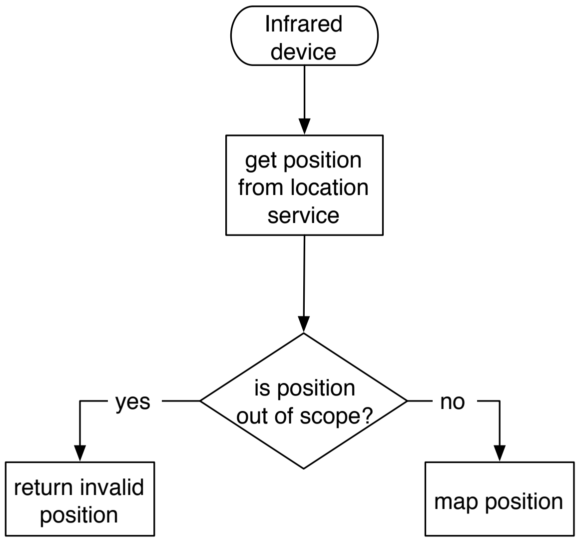

Figure 5.4: Position query of an infrared device

location service for the position of the associated node. Then it queries the location model, in order to map the position into symbolic coordinates. If the position is not defined in the location model, an invalid position identifier will be returned. Otherwise, the position given in symbolic identifiers will be returned.

5.2 Spatial Model

The spatial model reflects the environmental aspects of the simulation and the spatial constraints resulting from the physical environment. Modeling these constraints leads to a more realistic network simulator. Like the mobility engine, it is comprised of two major models, the *environmental model* and an arbitrary number of *location models*. Since more complex *radio propagation models* also take spatial constraints into account, they are mentioned briefly for the sake of completeness

5.2.1 Environmental Model

When simulating mobile ad hoc networks, we have to take care of spatial constraints in order to get more realistic results. Each simulation is restricted to a simulation area. The environmental model contains the spatial constraints of the simulation area. Figure 5.5 illustrates our concept which we describe below.

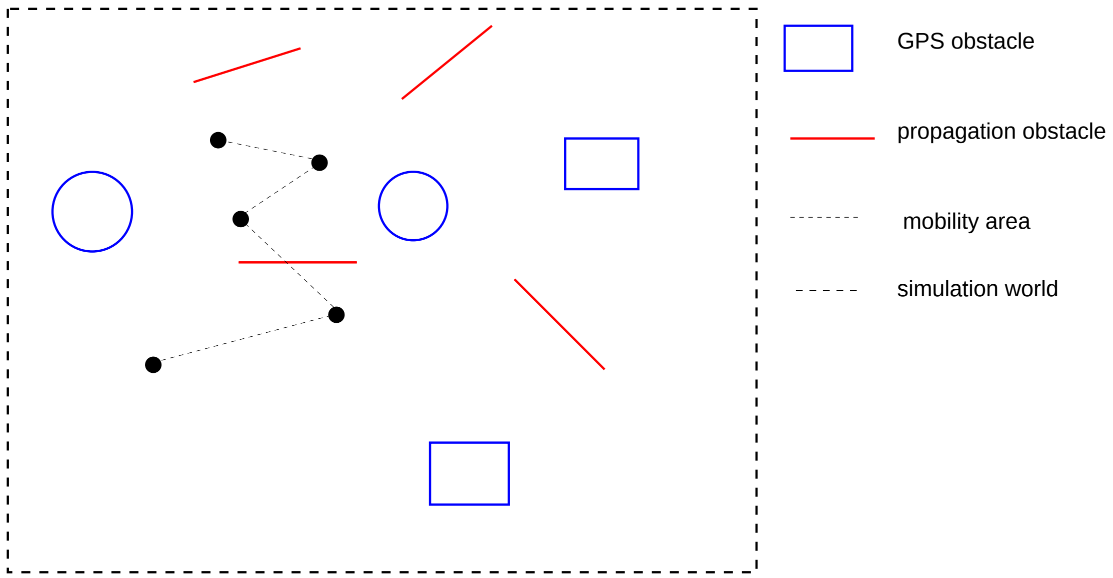

Figure 5.5: Environmental model: conceptual view

The environmental model defines a default area in which the nodes are placed, the "simulation world". For the sake of simplicity, the figure is shown in two dimensions, but in Cubus we also support three dimensions. In the simulation world, all kinds of mobility areas can be embedded. For instance, one can model graphs, rectangles or circles. These areas can be defined in two or three dimensions. They are stored in a *container class* which offers interfaces to retrieve areas of a specified type and to add arbitrary areas. Note that the simulation world itself is a mobility area, e.g a simple rectangle. It can be seen as the default mobility area without any obstacles.

When modeling radio propagation, realistic propagation models take care of obstacles like buildings and trees. The propagation model checks if two nodes are in line-of-sight of each other, i.e. if there is no obstacle between them. If nodes are not in line-of-sight, they may not be able to communicate. We model propagation obstacles as simple lines or plains in two dimensions or three dimensions respectively. They are also placed in the simulation area.

Furthermore, Cubus supports GPS devices. To model these devices as realistically as possible, we have to provide means to simulate areas, where the reception of a GPS signal is not possible. We model GPS obstacles as rectangles (cubes) or circles (spheres). These GPS obstacles are also embedded in our environmental model. If a GPS device receives a request for the position of a node, it first retrieves the current position from the location service and then checks whether the position is contained in a GPS obstacle. If the position is contained within a GPS signal, the GPS device returns an invalid position.

Besides the constraints, the environmental model contains all available location models. We provide consistency, by ensuring that every position mapped by a location model is represented in the environment. Consider a symbolic location model which maps a geometric coordinate into the symbolic identifier:
"Informatikneubau/zweiterStock/AbteilungVerteilteSysteme". The geometric representation of this building should be modeled in the simulation area.

The environmental model provides a unique interface for all the different containers. By providing this common interface, it is possible to add new spatial constraints to the environmental model. If a new constraint is created, the user has only to create a new container which holds all components of this kind. Additionally, we are able to dynamically load environmental models since they all provide the same interface. Hence, we are able to design very complex models as well as simple models which may contain just the simulation world without any obstacles and/or mobility areas.

Figure 5.6 shows an overview of the environmental model and its containers.

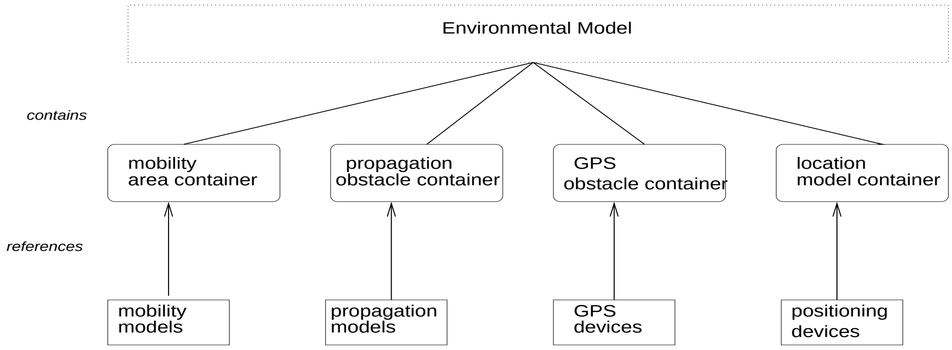

Figure 5.6: Environmental model overview

5.2.2 Location Models

A location model defines the transformation of positions, given in one format, into positions in another format. In the following, we list some of the requirements a location model must have to be able to transform a position.

- The transformation should be defined for every position in the simulation area. However, many location models have a restricted scope. Only positions located in the scope of the location model can be transformed. If a position is mapped from outside the scope, the location model returns an invalid position. As an example, consider a local geometric reference system which has the scope of a building. If a node is located inside the building, a transformation can be realized. Otherwise, the transformation is not possible and an invalid position is returned.
- We have to provide consistency, since the returned position must exist in the environmental model, e.g. the symbolic coordinate "Informatikneubau" should have a corresponding spatial element.

In Cubus, we support *geometric* and *symbolic* location models. With respect to geometric location models, we distinguish between global and local geometric models. Global geometric models contain the information to transform the position of a node into a global reference system, e.g. WGS84. Local geometric coordinate system contain the information to transform a position into a position of a local coordinate system. Symbolic location models map global positions to symbolic identifiers.

Location models are embedded in the environmental model to provide consistency. Each position in the format of the location model is represented by a coordinate in the simulation area. Every location model is defined with a scope which is either the total simulation area (e.g. a global geometric location model), or is a restricted scope (e.g a local geometric model with a scope of a building). Furthermore, each positioning device has a reference to its corresponding location model.

Figure 5.7 depicts the workflow of a location model's transformation. First it is checked, whether the position is inside the scope of the location model. If the position is outside the scope, an invalid position is returned, else the transformation can be realized.

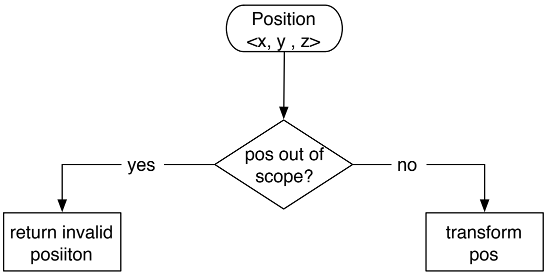

Figure 5.7: Transformation process

5.2.3 Radio Propagation Models

A radio propagation model determines the propagation of transmission signals and thereby the set of nodes which are in the transmission range of a sending node. Like mobility models, they are defined in C++ classes.

The radio propagation models in Cubus should mimic signal propagation as realistically as possible. The more complex a model is, the more computationally intensive it is. Therefore, we should also support coarse-grained models, which simplify the real world. In Cubus, propagation models provide *transmission range queries*, *path loss queries*, and *signal strength queries*.

A *transmission range query* determines all nodes that are in transmission range of a sending node. In order to build the radio interface list, the medium requests all nodes from the location service which are in a specific geographic range. The list contains all nodes which are potentially able to communicate. Secondly, the propagation model computes all nodes which are in transmission range using its underlying model.

A *path loss query* calculates the path loss of an outgoing signal, whereas a *signal strength query* calculates the resulting strength of a signal according to the path loss.

The propagation models are associated with a medium. To be able to compute the position of a node, they hold a reference to the location service. Furthermore, they hold a reference to the environmental model. Hence a model can determine, whether two nodes are in line-of-sight of each other by computing, if the line-of-sight intersects with any propagation obstacles.

All propagation models are derived from a common base class. By this, we provide a common interface for all propagation models. They can be exchanged during runtime of the simulation.

5.3 Summary

In this chapter, we have examined environmental aspects like mobility of nodes and radio propagation. We have introduced the *mobility engine* and the *spatial model*. The mobility engine deals with all aspects of mobility, whereas the spatial model represents the physical world, in which algorithms are run.

The mobility engine is comprised of the *location service* and *mobility models*. The location service can be seen as an interface between the mobility models and the rest of the system. It provides position information by applying mobility models. Mobility models define the movement of their associated nodes.

The spatial model is comprised of the *environmental models* and *location models*. The environmental model is the repository for all kinds of spatial constraints. It contains mobility areas, propagation obstacles and GPS obstacles. Any kind of spatial constraint can be added, as the environmental model provides a common interface to the user. Moreover, the environmental model contains location models. These models map positions, given in one format, into positions of another format.

6 Design of the Demonstrator Engine

This chapter describes the components which provide interactivity and analysis capabilities for our simulator. We first examine which problems may arise when supporting user interactivity and how we can solve them. Afterwards we discuss in detail the demonstrator engine. It is comprised of the *interactive engine* and the *analysis engine*. The interactive engine provides interactivity for the user. The analysis engine is responsible for the storage of log events.

6.1 Interactivity and its Problems

Most simulators provide a graphical user interface which animates the state changes of the system. They often lack the capability to let the user interact with the running system. Cubus provides a mechanism to support the retrieval and setting of parameters. However, we have to face some problems which arise when supporting interactivity.

First, when supporting interactivity, we have to cope with inconsistencies, that may arise in the system. Consider the scenario depicted in Figure 6.1.

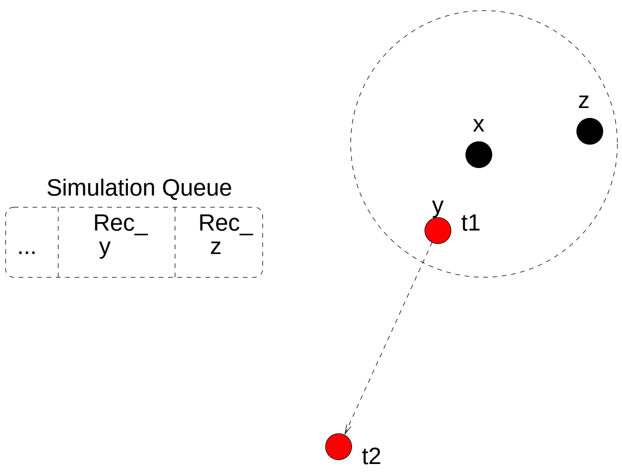

Figure 6.1: Inconsistency scenario

Node x has sent a packet to nodes y and z at time *t1*. In the simulation queue, a receive event for y is already scheduled. The user changes the position of node y at time *t2*, e.g. by dragging it in the graphical user interface to the new position. The receive event will be handled, though the node is not in the transmission range anymore. Note, that this situation does not affect send events, since the neighboring nodes are computed when the send event is handled.

The question arises how this inconsistencies can be solved. One solution is to inform the user, when he changes the position of a node and there are still receive events scheduled for the node. The user then can discard the action. Otherwise, the simulation results may vary from reality. Another idea is to define a change position event for the specific node. This event has to be scheduled after the receive event. But this does not prevent the system from scheduling incoming receive events. These receive events are based on the old state of the node. In order to prevent the creation of events for this node which are based on old values, one has to block incoming events after a change position event is scheduled. However, this leads to a slower simulation. Therefore, the first solution fits our needs best, since we are able to detect inconsistencies without slowing down the simulation.

Another problem is the synchronization between the graphical user interface and the simulation. If the simulation is very fast, i.e. the simulation time is greater than the wall clock time, the graphical user interface might not react fast enough to the simulation, since graphical updates can take very long. This can become a problem, in particular, when the simulator runs on a different machine than the graphical user interface. If the user wants to change simulation parameters, he might not have the current state of the simulation.

To solve this problem, the simulation control has to synchronize with the graphical user interface. One could define an interval after which the simulation pauses and the graphical user interface updates itself.

In the next section, we discuss our interactive engine and how we deal with the above mentioned problems.

6.2 Interactive Engine

As we have seen in Chapter 4, most network simulators lack support for user interactivity. Since Cubus is not only designed as a simulator, but is also supposed to be used in lessons as a demonstrator tool for data management algorithms, we need to provide animation and interaction capabilities. The user should get and set parameters of the simulation and should be able to exchange components on-the-fly, e.g. mobility models and algorithms.

In Cubus, the interactive engine is the component which provides this capability. Figure 6.2 depicts an overview of the interactive engine. The interactive engine provides a common

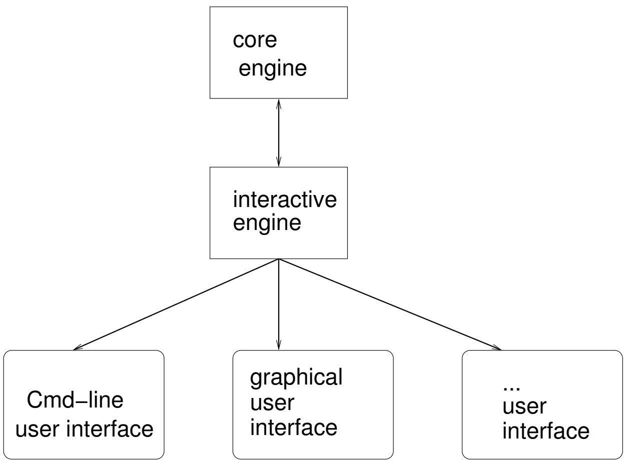

Figure 6.2: Overview of the interactive engine

interface for various kinds of user interfaces. Besides graphical user interfaces, Cubus also supports command-line user interfaces whose interface is similar to the GNU Debugger GDB [GDB04]. An arbitrary number of user interfaces can be attached to the interactive engine, but at any time only one of them is active. All types of user interfaces need to provide the following mechanisms to control the simulation:

- Get the current simulation time
- Pause the simulation
- Resume the simulation
- Stop the current simulation run
- Set a breakpoint at a specific simulation time or when reaching a predefined event.

Breakpoints can be considered as events with a precondition, e.g. when node x is sending a message. When a breakpoint is reached, the simulation is paused. For the realization of breakpoints we use the *publish-subscribe* mechanisms provided by the core engine. Consider a breakpoint for a send event of node a. The breakpoint is a subscriber for send events. When a send event is taken from the simulation queue, all subscribers are informed. They check, if their associated condition is true. For example, the breakpoint for node a checks, whether the send event comes from this node. If the condition is true, the core engine is paused and the current user interface will be informed that the breakpoint is reached.

While using the command-line user interface is more suitable when the simulation needs to run fast, a graphical user interface is preferred for debugging and teaching purposes. It needs to support the retrieval of system parameters. Some examples, which actions a graphical user interface might support, are the following:

- Exchange algorithms on-the-fly
- Create a new mobility model
- Replace a mobility model with another existing mobility model
- Change parameters of a node

In order to be able to set and retrieve parameters, we have to provide information about all parameters in the system.

6.2.1 Parameters

A parameter is a property of a component, e.g. *maximum speed* of the Random Waypoint Model. Each object of a component has two types of parameters: *Predefined parameters* are defined for all objects of a class, e.g a global unique identifier for each object. *Custom parameters* can be freely defined by the user. Every custom parameter in our system has the following entries:

- *name*: the name of the parameter, e.g. "queueSize"
- *value*: the type of the parameter , e.g "integer"
- *range*: maximum and minimum value, the type can take, e.g "1...20"
- *stepSize*: size of an increment or decrement step to this parameter, e.g. increment or decrement a parameter in steps of one

In order to store and manage parameters of a component, [Molt04] introduced a basic *container* class. Component specific containers, called *repositories* inherit from this super class for each component in the system with parameters. Hence, we have repositories for each dynamically loadable components, like mobility models and algorithms.

In Figure 6.3, a detailed view of the algorithm repository is shown. The repository assigns parameters to the corresponding algorithm. The parameters of a new object of a specific

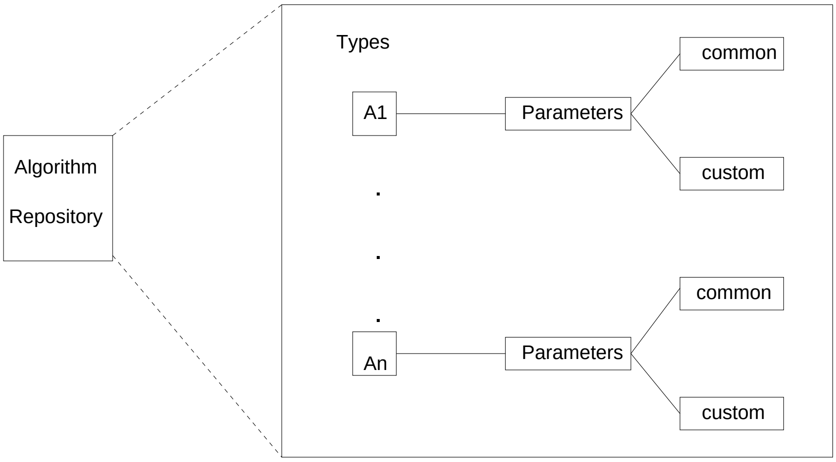

Figure 6.3: Internal structure of algorithm repository (adapted from [Molt04])

type, e.g. algorithm, are stored in the repository. The common base class defines generic *get* methods which can either retrieve parameters with a given name or all parameters from a component, e.g. all parameters of the Simple Flooding algorithm. The system is extensible, i.e. new repositories can be added. To provide a common view on repositories, we introduce the *parameter repository*.

6.2.2 Parameter Repository

The parameter repository provides a common interface to all repositories, by forwarding parameter requests to the appropriate repository. In Figure 6.4, an overview of the architecture is depicted.

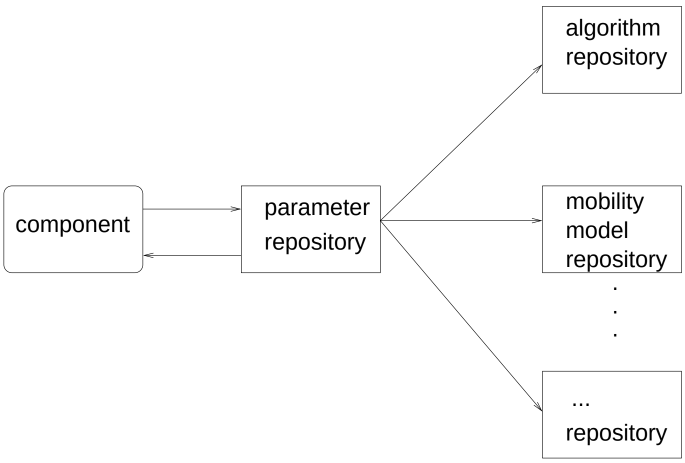

Figure 6.4: Parameter repository with associated component repositories (adapted from [Molt04])

In our design, we use the parameter repository as a name service for parameters. Therefore, we have to provide a naming convention for parameter requests.

A request is built according to the format of a Universal Resource Identifier (URI), since URIs have the advantage of providing human-readable format. All parameters which are located on the path of the URI are returned. The entry point is the root named `cubus://`. Consider two examples:

- `cubus://algorithms/`: This request returns all available parameters of algorithms

- `cubus://algorithms/SimpleFlooding` returns all parameters of the Simple Flooding algorithm

In the parameter repository, a request will be mapped to the appropriate container by calling the container's generic *get* method. The URI is set up by the user within the graphical user interface.

The advantage of our system is that users do not have to recompile parts of the system when they add a new component with parameters. Moreover, by knowing the internal structure of objects, we are able to dynamically load components during runtime with appropriate parameters. The question now arises how we can retrieve or set values of an instantiated object.

In object-orientated languages, the object usually has *get* and *set* methods for all its attributes. In order to call these methods, the user interface has to know their name. When having a lot of objects, it is very memory-consuming to store all names of all available methods. Also, parameters of objects may change or be removed, including their *get* and *set* methods. The user interface then would have to be informed about the changes. In Cubus, we have chosen another approach.

All objects including their parameters are derived from the common base class `CubusObject`. This class defines generic *get* and *set* methods which are overriden by each derived class. The user interface then just calls the generic *get* or *set* methods, with the name of the specific parameter as argument.

For the interaction with the running simulation, a graphcial user interface is needed. In the next section we examine how to support animation and user interactivity in our Cubus system.

6.2.3 GUI Support

One task of a graphical user interface is to visualize the state changes in the system, e.g. location of nodes and the packet flow. It must be ensured that it always displays the current state of the simulation. In Cubus, we follow the approach of GTNetS, which is described in Chapter 4 on page 34.

The graphical user interface is derived from the event handler base class and therefore can handle events. It schedules update events in specific time intervals. When an update event occurs, the simulator is paused and control of the simulation is handed to the user interface. The user interface updates every component it is interested in, e.g. the location of all nodes. After all processing is done and the user has finished the task, a new update event is scheduled. Figure 6.5 depicts the sequence, when an update event t is handled by the user interface.

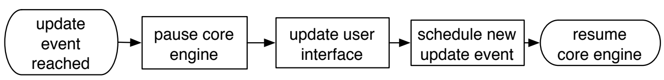

Figure 6.5: Update event handling by the graphical user interface

The update events are called *sync points*, since the user interface synchronizes with the core engine each time a sync point is reached. The interval between two sync points can be chosen by the user. If it is very small, few events have happened in between, if the interval is very large, many events have happened in the simulation. Hence, by choosing the interval, the user chooses the animation rate.

We want to optimize the Cubus system, and therefore provide capabilities to put the interactive engine on another machine. When handing control to the user interface, we must ensure that no events are processed and scheduled until the user wants to resume the simulation. Therefore, when the interactive engine receives an update event, the core engine is paused. After all processing is done in the interactive engine, the core is resumed and can go on with the event processing.

The only drawback of our concept is the restriction of the user interaction. The user interacts with the system when a sync point is reached and control is given to the user interface. The notion of sync points provides the user interface with the current state of the simulation and control over the animation speed. However, inconsistencies can occur which we have discussed in Section 6.1, especially, when the user changes the position of a node, for which a receive event is already scheduled.

In Cubus, the change of a node's position is defined by the associated mobility model, since some mobility models do not allow the change of the position, e.g consider a group-based mobility model, in which a node is set away from its group. The mobility model designer has to detect and handle the inconsistency. Usually, the mobility model will query the core engine, if there still are receive events scheduled for the specific node. If there are events scheduled, the mobility model will send a warning to the system.

When running the simulation with a command-line user interface, the user is interested in the traces of the simulation run. The analysis engine is responsible for the storage of these traces. We discuss the analysis engine in the following section.

6.3 Analysis Engine

The analysis engine extends our system with logging and analysis techniques. It stores the logged events either in text files or in a relational database.

Cubus uses *log events* for logging which are derived from the event base class. The designer of a component decides which data is logged by the component and which log level is the default. The log level decides whether a log event is triggered. For ease of use, the user can also define a global log level in the configuration file which is valid for every component, unless it is overriden with another value.

Log Events are put in the *system event queue* of the core engine which runs asynchronously to the simulation. They are handled by the analysis engine which is the handler for all log events that occur in the system. The analysis engine runs asynchronously to the rest of the Cubus system. It has its own event queue, where the log events are put, after they have been removed from the system event queue. As a result, the control is given back as quick as possible to the core engine.

When processing the log events, the analysis engine writes them to a storage location. In addition to text files, we also support the storage of log events in a relational database. Note, that at each point of time, there is only one repository active. Figure 6.6 depicts the architecture of the analysis engine.

When storing log events in a relational database, we have to consider some aspects: Every type of log event is represented by its own table in the relational database. The log event itself includes the entries of a row. In Cubus, we define a common output format for each log event, which simplifies the storage of log events in a relational database. The base class of log events defines a vector of key-value pairs, where the value is stored in the column

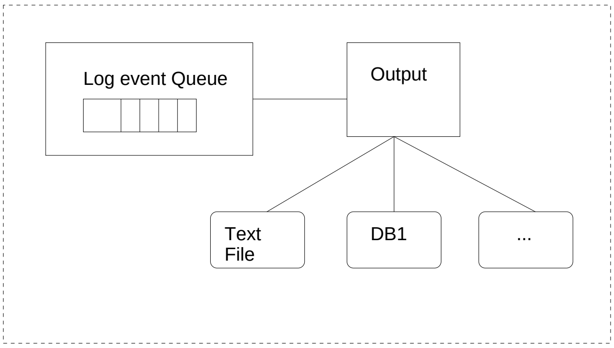

Figure 6.6: Analysis engine: architectural overview

named key. Hence, the user does not have not to compile parts of the analysis engine, when adding new log-events.

In order to provide information to the system about the internal structure of a log event, the user has to define them in a configuration file which is parsed during startup of the simulation. The configuration files have the following structue:

```
Name Logevent{
Name: Type;
....
}
```

`Type` is every data type a relational database supports.

Each row in the database table contains entries for the current simulation run which is set at the start of the simulation, and the simulation ID. The simulation ID identifies a simulation and the most important parameters, e.g. number of nodes and the topology.

The interactive engine can also create log events. For example, when a user changes a parameter of some component, a *change event* is generated. The change event contains the current simulation time, the identifier of the component, the name of the parameter being changed, the old, and the new value.

The analysis engine is not only a handler for log events, but it can also be extended with analysis capabilities. Users can augment it with histogram functionality. Moreover, since we support relational databases, we could plug in classes which read tables out of the database and forward them to an external plotting tool, e.g. Gnuplot.

6.4 Summary

In this chapter we have discussed how to support user interactivity and analysis capabilities. We have outlined the problems which may arise when supporting interactivity in simulators by means of a graphical user interface. It must be ensured that the user always has the current state of the simulation and that the user interface synchronizes with the simulation control. We have examined an example of an inconsistency which may arise when events are scheduled for an old state of the system.

Next, we have discussed the design of the demonstrator engine. The demonstrator engine provides interactivity and analysis capabilities for our system. It consists of the *interactive engine* and the *analysis engine*.

The interactive engine supports various user interfaces, among them a command-line user interface and a graphical user interface. A command-line user interface is used when running the simulation as fast as possible, whereas a graphical user interface is used for debugging facilities. A graphical user interface is as an event handler for update events. These update events are called *syncPoints*, since they synchronize the graphical user interface with the core engine. Interaction is only possible, when a graphical user interface handles such an update event.

The analysis engine stores the log events in text files or a relational database. Each log event is represented by a row in the database. In order to support the storage of events in relational databases, the base class of all log events defines an output format which contains all data of the event as key-value pairs. The keys are the names of the columns of the associated rows, whereas the values contain the logged data.

7 Implementation

In this chapter, we describe the implementation of our prototype. First, we discuss briefly the implementation of the components which were designed in [Molt04]. The realization of the mobility engine, spatial model and demonstrator engine is examined in the second part of this chapter. We conclude with a discussion of the extensions we have built into to the components of [Molt04].

7.1 Implementation Basics of the Previous Thesis

Our protoype is implemented in C++ [Oual02, PrKP02]. Care was taken to stay as close as possible to the ISO C++ standard. For efficiency reasons, parts of the implementation are realized by STL templates [Sili04]. To support dynamic loading of components, wrapper classes have been defined, as described in [Nort00]. Two kind of files are distinguished: *scenario files* and *configuration files*.

The user defines the whole simulation and its topology in the scenario file. The scenario file is passed to the simulator at startup of the system and is processed by the parser to build the corresponding components.

Configuration files contain information about a component's internal structure. This is needed in order to be able to dynamically load an object. Each dynamically loadable object has a corresponding configuration file which provides the parameters of the object. The configuration files are parsed at the start of the simulation. The parameters are stored in *repositories* for later retrieval.

The parser was generated with the tools *flex* and *yacc* [LeMB92] which generate LALR(1) parsers. First, the parser reads the configuration files and stores all parameters in the appropriate repositories. It then parses the scenario file and builds the scenario. After that, the simulation is started. Figure 7.1 depicts the sequence of steps at simulation startup.

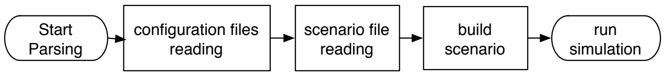

Figure 7.1: Simulation startup process (adapted from [Molt04])

Figure 7.2 depicts the architecture of the Cubus system. The *simulation engine* and the *algorithms* have been implemented in [Molt04]. A more detailed description can be found in [Molt04] on page 79. In the following, we examine the realization of the components described in this thesis.

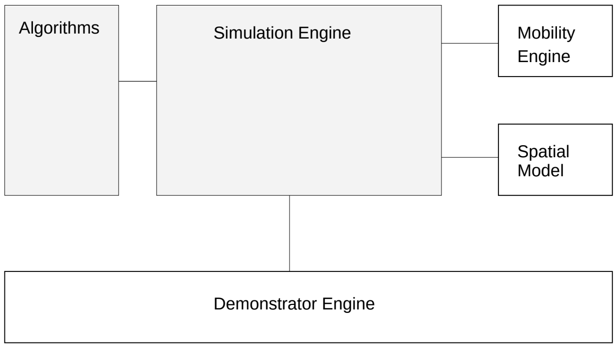

Figure 7.2: Conceptual view of the mobility engine

7.2 Implementation of the Mobility Engine

Figure 7.3 depicts the conceptual view on the mobility engine we have outlined in Chapter 5. Each mobility model is associated with a list of nodes. The location service can be seen as an interface for other components interested in the position of nodes.

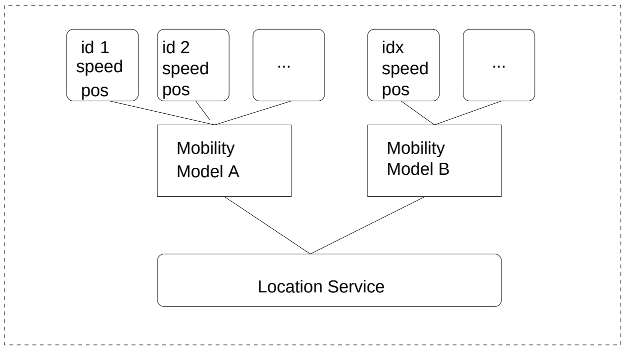

Figure 7.3: Mobility engine: implemented components

7.2.1 Mobility Models

The common base class for all mobility models is derived from the abstract base class `CubusObject` which defines generic *get* and *set* methods for the setting and retrieval of parameters. The `get(char* parameterName)` method returns the value of the given parameter. The method `set(char* parameterName, void* value)` sets the value of parameter `parameterName` to the given value.

Each mobility model has a list of its associated nodes. This list contains the node's globally unique ID and its position and speed. Each mobility model can augment this data structure with additional information, e.g. the *Random Waypoint Mobility Model* stores a destination for each associated node.

The movement of a node is recomputed, as soon as the function `computePos(Node*, Time)` is called from another component.

There are two possible ways of instantiating a mobility model:

1. The user can hand over the name of a mobility area on which the nodes of the model move.
2. If there is no mobility area given by the user, the model is associated with the default area of a specific type, e.g. graph.

The mobility model defines methods to return the speed and position of a node and to remove and add nodes. Since some mobility models might not allow the removal or addition of nodes, a boolean variable dictates whether the model supports this kind of operation.

We have implemented the *Static Model* which models static networks in which nodes are immobile and the *Graph-Based Model* [THBS+02]. The *Random Waypoint Model* has been adopted from the GTNetS implementation. All these models are compiled as shared objects and can be loaded dynamically during the runtime of a simulation.

7.2.2 Location Service

The location service is implemented as a linear list which contains references to all nodes of the system. Each node holds a reference to the mobility model it is assigned to. When a position query for a specific node occurs, the location service hands over this request to the specific node. By calling the corresponding `computePos(Node*, Time)` methods of the associated mobility model, the current position of this node is returned.

In a range query, the range is represented as a circle or sphere with a given radius in 2D or 3D respectively. The linear list which contains all nodes is traversed and the position of each node is computed. When the distance between the current node in the list and the sending node is less the radius, they are assumed to be in detection range. The decision whether two nodes can communicate is determined afterwards by applying a radio propagation model.

7.3 Implementation of the Spatial Model

7.3.1 Environmental Model

The environmental model stores mobility areas, propagation models and GPS obstacles in STL maps [Sili04]. It implements the methods `registerElement` and `removeElement` which allow the user to add and remove arbitrary components.

It supports three kind of methods for mobility areas. The method `getMobArea(char* name)` returns the mobility area with the given name. The method `getDefaultArea(char* type)` returns the default area of the given type. The method `getMobArea(char* type)` returns all mobility areas of a given type, e.g. all graphs.

Propagation and GPS obstacles can be obtained by providing their globally unique identifier. The environmental model provides access to the corresponding maps for the propagation models and GPS devices. Whether positions are in line-of-sight of each other or if there is an obstacle between them is computed in the propagation model.

The current structure of the environmental model can easily be extended to support any kind of component, e.g. location models.

Environmental models can be loaded dynamically. Currently a simple environmental model is implemented, using a rectangle and a graph as mobility areas.

7.4 Implementation of Radio Propagation Models

All radio progagation models are derived from the abstract base class `propagation`. This base class has references to the location service to query for positions of nodes and to the environmental model to be able to retrieve propagation obstacles. Each propagation model has a reference to the medium it is attached on. The base class defines three methods that have to be overwritten by each derived propagation model.

`TRQuery(WirelessInterface* , interfaceList)` traverses the *interfaceList* which contains all nodes that are in the detection range of the sending node. The nodes which can receive the signal, are computed by applying the underlying propagation algorithm.

The method `inTRRange()` determines whether a node is in the transmission range of the sending node. The method `getPathLoss()` calculates the path loss of a signal from one node to another node.

7.5 Implementation of the Demonstrator Engine

7.5.1 Analysis Engine

All log events are derived from the common base class `LogEvent`. This class defines a generic output format for all parameters which are logged by the event. The format is a map of <key, value> pairs, where the key is the name of the parameter.

We have defined an abstract base class `Output` which offers the method `logEvent(Event*)`. This method writes the content of the log event to its appropriate specific storage location, e.g. a relational database. All derived output classes have to overwrite this method. The current version supports two different kinds of output, text files and output in a relational database.

To reduce the I/O overhead for the system, the analysis engine is equipped with its own event queue which runs asynchronously. This is implemented using the Posix Thread Standard [Labo04].

7.5.2 Interactive Engine

The current version supports two user interfaces:

1. The command-line user interface. It supports no interactivity and is supposed to be used when the simulation should run as fast as possible, e.g. when evaluating multiple simulation runs.
2. The graphical user interface, implemented in Qt [Trol04]. It is derived from the base class `EventHandler` and handles update events. Each time an update event is reached, the core engine is paused and control is handed to the user interface. The user interface then updates the positions of all nodes. Afterwards, a new update event is scheduled.

7.6 Extensions to the Components of the Previous Thesis

In the following, we outline the extensions to the components developed in the previous thesis in order for them to support our new requirements. We have extended the grammar which defines the structure of configuration and scenario files. We also included the parameter repository into the core engine.

```
cmobilityModel {
name = ...;
type = ...;
location = ...;
property{
        name = parameter1;
        min = 5.0;
        max = 5.0;
        step = 1.0;
        editable = true;
        default = 5.0;
        };
        ...
};
```

Listing 7.1: Mobility model configuration file

7.6.1 Extension of the Grammar for Scenario and Configuration Files

The grammar for the scenario and configuration files is written for the yacc parser generator and defines the structure of the configuration and scenario files. We have extended the grammar to add support for mobility models and log events. Figure 7.4 depicts which files are supported in the current version of Cubus.

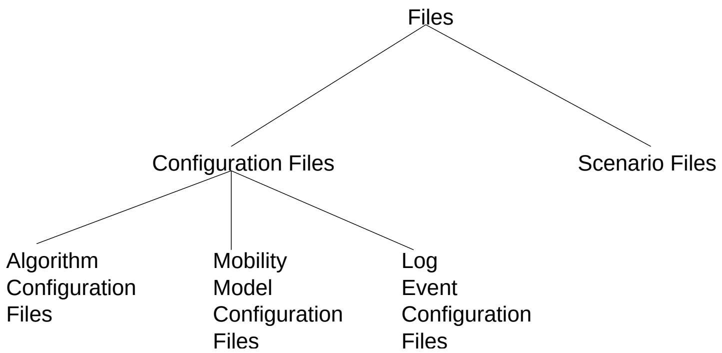

Figure 7.4: Supported file types in Cubus

In the following, we briefly discuss the configuration files for mobility models and log events and the extensions to the scenario files.

Configuration File for Mobility Models

Since mobility models should be dynamically loadable, they have to provide information about their internal structure to the system. The parameters of a mobility model are defined in a configuration file which is located in the directory `mobilityModels`. Each mobility model has its own configuration file.

The configuration file contains the custom parameters and their default values. When a mobility model is instantiated, the parameters are set to their corresponding default values. Moreover, all configuration files share the three properties `name`, `type` and `location` which identify the mobility model and provide the information, where the compiled shared library of this model is located. In Listing 7.1 a basic configuration file is given.

```
Name logevent {
        parameter 1: type1;
       parameter2: type2;
       ...

 };
```

Listing 7.2: Log event configuration file

Configuration Files for Log Events

Each log event supports the common output format of a vector with key-value pairs, where the key is the name of the data being logged. The data is stored in `void*` pointers. For the storage of the data in a relational database or a text file, the system has to retrieve the type of the data. Therefore, the user defines the internals of each log event in a configuration file. This file is stored in the directory `logEvents`.

Each configuration file contains the name and the type of the data to be logged. In Listing 7.2 a rudimentary example of a log event is depicted.

Extension to the Scenario File

We have extended the grammar for the scenario file to support the declaration of an environmental model and an arbitrary number of mobility models. Each scenario file must define an environmental model which determines the number and kind of mobility areas, propagation and GPS obstacles.

An arbitrary number of mobility models can be loaded into the systems. Their declaration is not mandatory. If there is no mobility model given in the scenario file, all nodes are assumed to be immobile.

The user may assign nodes to specific mobility models. If there is no association between nodes and mobility models, the nodes are assigned to the Static Model, i.e. all nodes are immobile. A basic scenario file is given in Listing 7.3.

In the next subsection, we discuss the extensions we have built into the core engine.

7.6.2 Extensions of the Core Engine

The following additional concepts have been added to the core engine:

- The core engine now has additional references to the location service to request the positions of nodes and to the environmental model. Moreover, it includes all instantiated objects in the simulation and all containers that store the parameters of objects.
- The containers are implemented using STL maps with the name of the object they store as the key, e.g. `SimpleFlooding` denotes the Simple Flooding algorithm inside the algorithm container. The container base class defines the method `getParam(char* name)` which returns the information about the parameters of the component denoted by `name`. Additionally, all instantiated objects are stored in STL maps for fast retrieval.

```
cenvironmentalModel {
        EnvironmentalModel  location model1;
};
cinterfaces {
        Interface location inf1;
};

cmedium {
        Wlan location med1;
};

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

cregister(inf1, a0);

sharedLink{
        med1 -> node1(inf1);
};

cmobilityModel{
  RandomWayPoint "../examples/RandomWayPoint.so" mob1;
  area = DefaultRectangle;
};

cregister(node0, mob1);
```

Listing 7.3: Extended scenario file

- The parameter repository is realized within the core engine. The method `getParameterInformation(cubusType t, char* name)` returns all parameters of the component denoted by `name`. The appropriate container to which this request is handed to, is identified by the enumeration type `cubusType`.
- Besides the extension we have built in the core engine, we have changed the implementation of the system event queue. The system event queue is implemented as a thread which can run asynchronously. We have implemented the thread based on the Posix Thread Standard [Labo04] since it is a well-known standard which provides interoperability for our system. The synchronization between the system event queue and the simulation queue is done through semaphores.

7.7 Summary

In this chapter, we have examined the implementation of our prototype. We first have discussed the implemenation basics of the previous thesis. The system was implemented in C++ due to efficiency reasons. The dynamic loading of components was realized by providing wrapper classes according to [Nort00]. We distinguish two kinds of files, *configuration* and *scenario* files.

Before the start of a simulation run, the configuration and scenario file are parsed. A configuration file contains the parameters of an object that is dynamically loadable, e.g algorithms and mobility models. The parameters are stored in container classes. The scenario file contains the description of the simulation topology, i.e. which nodes, media or algorithms are used and how they are connected.

We have extended the scenario file to support the declaration of an environmental model and mobility models. We have also supplemented the grammar to let the user register specific nodes with a specific mobility model.

The core engine now contains all parameter containers and implements the generic *get* and *set* methods in order to forward parameter requests to the appropriate container. Additionally, the *system event queue* now runs asynchronously and is implemented as a Posix Thread.

Mobility models are derived from a common base class and therefore have a common interface. Since they need to be loaded dynamically, the parameters of a mobility model are stored in a configuration file.

The environmental model can be regarded as a container which contains mobility areas, propagation obstacles and GPS obstacles. All obstacles are stored in STL maps for easy retrieval. The environmental model returns a reference to the propagation obstacles for radio propagation models, since the computation, whether two nodes are in line-of-sight takes place at the propagation model. Moreover, it supports the request for mobility areas of a given name or type. All environmental models are derived from a common base class which provides methods to add and remove arbitrary components.

The interactive engine contains two user interfaces. A command-line user interface which is used to run the simulation as fast as possible, and a graphical user interface which is implemented in Qt. As soon as an update event is reached, the core engine pauses and control is handed over to the graphical user interface which updates the positions of all nodes in the system.

The analysis engine includes its own event queue, which runs asynchronously to the rest of the system. Log events can be stored either in a text file or in a relational database.

Therefore, we implemented a base class `Output` which offers a common interface for the logging of events in an arbitrary location.

8 Evaluation

In Chapters 5 and 6, we have examined the design of our extensions to Cubus in detail. In this chapter, we evaluate the behavior of our extensions in a running simulation and compare the results with those of [Molt04].

8.1 Motivation and Constraints

The main goal of the evaluation is to determine if our extensions had an impact on the simulator's overall performance or behavior by comparing them to the results gathered in [Molt04].

In order to be able to assess our results, we also compare them to other network simulators. GTNetS was chosen since we have adapted some components from it. NS-2 is used in the comparison since it is the most commonly used network simulator. One aspect we have to keep in mind is that comparing simulation runs on different simulators is always difficult. According to [CaSS02], the same scenario can behave in an entirely different fashion when running on different simulators.

Currently, a simple graphical user interface and an environmental model without propagation obstacles are implemented. Therefore, we are not able to evaluate every aspect of our extensions. For other aspects, e.g. user interactivity, we lack the appropriate metrics. Hence, we do not evaluate more complex environmental models and user interactivity.

8.2 Evaluation Metrics

We distinguish two categories of metrics. The first category comprises metrics which measure the performance aspects of our system. The *Setup Time* and *Runtime* metric belong to this category. The *Setup Time* metric determines how long the setup of a scenario takes. It is desirable to have a low setup time. The *Runtime* metric determines how long a simulation run takes in wall clock time. The goal for the Cubus system is to take less runtime than NS-2 and GTNetS.

The second category evaluates the behavior of the simple flooding algorithm and compares the results with runs on other network simulators. The *Latency* metric belongs to the second category. The *Latency* metric is defined as the average time needed for one packet to reach node n. With increasing number of nodes, the latency should also increase, since collisions get more likely and a node has to wait longer until it receives its first packet.

8.2.1 Parameters

We use the same setup as in [Molt04]. Our nodes are randomly placed in a 1000 m $\times$ 1000 m square which contains no obstacles. Each simulation run, the nodes are either all static (*static scenario*) or move according to the Random Waypoint Mobility Model (*mobile*

```
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 algorithm

scenario). Since there are no propagation obstacles, all pairs of nodes that are within transmission range of each other can communicate. The transmission range is set to 250 m and the Free Space Model is used for signal transmission.

Each node is equipped with the same protocol stack: A *Simple Flooding algorithm* and *application*, as well as a MAC-layer based on IEEE 802.11. The Simple Flooding algorithm was adopted from [Molt04] and is shown in Listing 8.1.

The sequence of a simulation run is as follows: A randomly chosen node starts with the transmission of a packet at the time given in the scenario file. The simulation stops when no more packets are transmitted and all reachable nodes have received the packet.

Table 8.1 summarizes the constant parameters of the simulation. The mimimum and maximum node speed and pause time are adapted from [CaSS02]. Each simulation scenario is run ten times and the results are plotted with the standard deviation, when appropriate.

Parameters	**Values**
Terrain size	1 km × 1 km
Node placement	uniform
Number of broadcasting nodes	1
MAC protocol	IEEE 802.11
Bit rate	2 Mbps
Radio propagation model	Free Space
Mobility Model	Random Waypoint
Min. pause [sec]	0
Max. pause [sec]	1
Min. speed [m/s]	0
Max. speed [m/s]	5

Table 8.1: Static parameters

8.3 Simulation Runtime

8.3.1 Setup Time

In [Molt04], the setup times of NS-2, Cubus, and GTNetS were compared in the case of the static scenario. The results are depicted in [Molt04], on page 88. The smallest setup time was reported for GTNetS which can be attributed to its handling of scenario configuration files: In GTNetS, the user specifies a scenario in C++ within the `main` method. Among the two simulators using external scenario files, Cubus clearly outperforms NS-2. This is due to the two-split programming model of NS-2 using OTcl and C++. Additionally, NS-2 precomputes characteristics of the network during the setup process.

When building our extensions into the Cubus system, it is important that the setup time does not grow heavily and hence degrades the results of [Molt04]. Figure 8.1 depicts the comparison between the setup time in the static scenario and the mobile scenario with an increasing number of nodes.

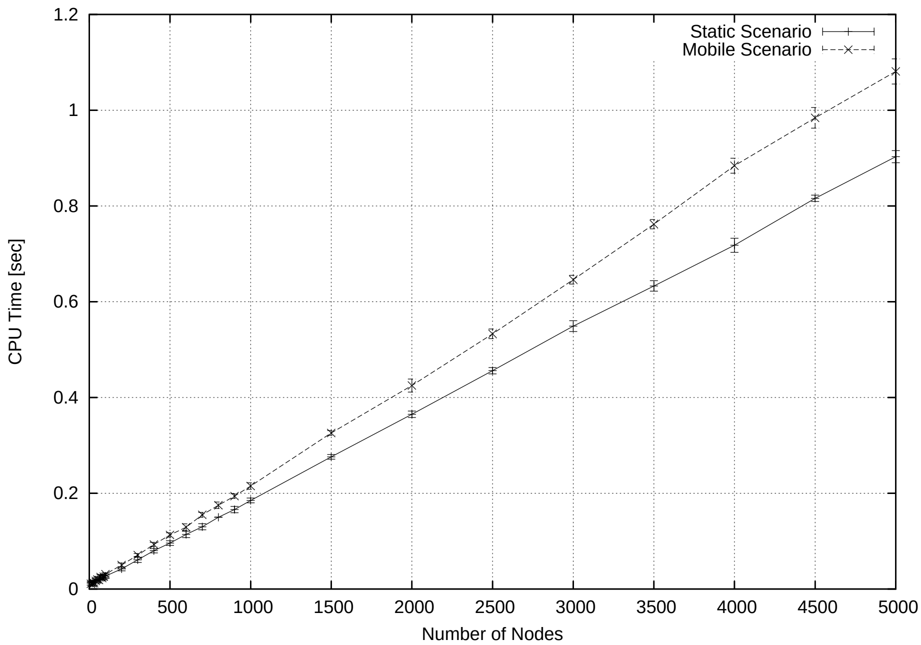

Figure 8.1: Cubus setup time comparison between static and mobile scenario

Clearly, the setup time in the mobile scenario still grows linearly. This is in accordance with the requirements of [AhSU86], that a good parser should not need more than linear time to read a file. The setup time in the mobile scenario is greater, since the definitions of the environmental model, the mobility models and the connections between nodes and mobility models have to be read additionally.

8.3.2 Runtime Comparison

After examining the setup time of our simulator, we measure the runtime behavior of Cubus. We compare our results with NS-2 and GTNetS. The scenario is described in the previous section. We increment the number of nodes to 100 in steps of ten nodes. We repeat each run twice, one run for the static scenario, the other for the mobile scenario, in which all nodes move according to the settings listed in Table 8.1.

Figure 8.2 summarizes our results with increasing number of nodes where all nodes are static. Due to its two-split programming model, NS-2 has the fastest increasing runtime

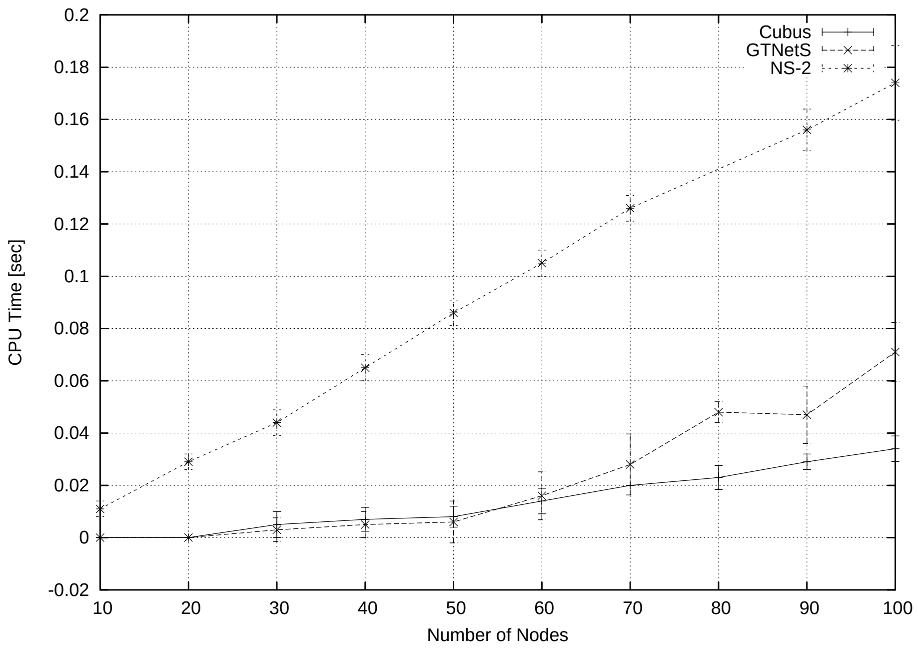

Figure 8.2: Runtime comparison with static scenario

of all simulators since method calls have to be dispatched to the appropriate C++ classes which requires more time with an increasing node number.

Figure 8.3 plots the runs of GTNetS and Cubus for a greater number of nodes. Up to 1000 nodes, Cubus and GTNetS behave similar to each other. For a greater number of nodes, the runtime of GTNetS increases much faster than the runtime of Cubus. This can most likely be attributed to the more sophisticated algorithms and protocols used in GTNetS. Further development of Cubus, such as adding more realistic wireless interfaces, is likely to increase the runtime in a similar fashion.

These results were also obtained in [Molt04], though the CPU runtimes are a factor of 10 higher than in our results. These differences occur, because the results of [Molt04] were measured on a different machine and the simulation was run in debugging mode. Nevertheless, the curves of both theses show similar behavior.

In Figure 8.4 the plots are shown for the mobile scenario, in which all nodes move according to the Random Waypoint Mobility Model. We can see that all simulators take longer in their runs than in the static scenario. This is caused due to the mobility of the nodes, requiring the positions of nodes to be calculated.

Again, NS-2 has the fastest increasing runtime. Like in the static scenario, GTNetS and Cubus show the similar behavior. Figure 8.5 plots the runtime of GTNetS and Cubus with larger scenarios, in which all nodes move.

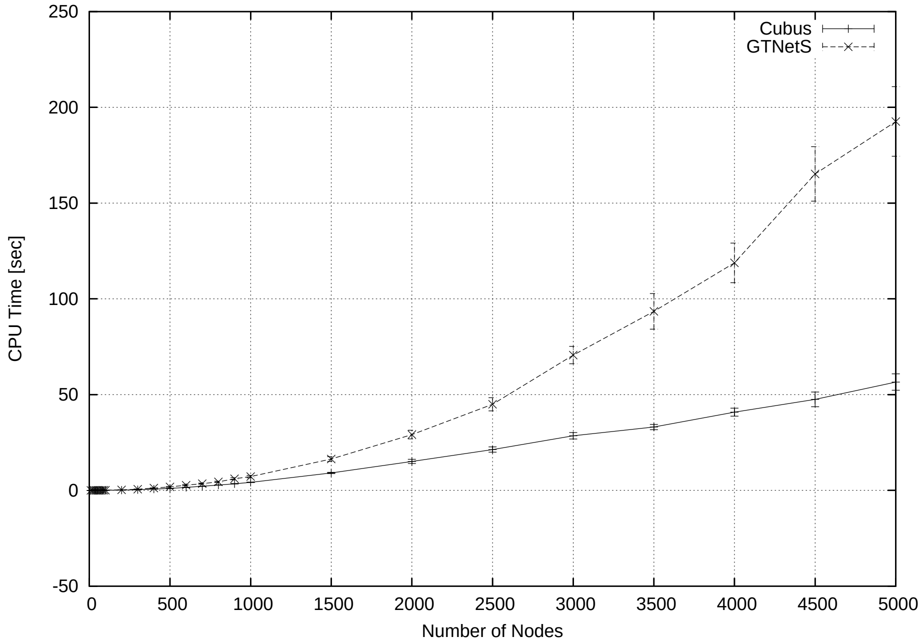

Figure 8.3: Runtime comparison with larger static scenarios

We have seen that Cubus shows a similar behavior to GTNetS in runtime, even if all nodes are mobile. The difference between the static and mobile scenario is tolerable. Hence we have shown that our mobility engine design not to degrade the behavior of Cubus.

An important point to mention is the development stage of our simulator. GTNetS and NS-2 contain more detailed algorithms and protocols. When adding more complex models to Cubus, we expect a degrade in the runtime. However, we still expect a linear order, since some approaches are more efficient in our system than in NS-2 and GTNetS, e.g. the mixed method- and event-based communication.

8.4 Latency

In the previous sections, we have considered the performance aspects of our Cubus system and have compared some of the results with NS-2 and GTNetS. [Molt04] has examined the behavior of the Simple Flooding algorithm by measuring the latency of the simulated network.

The latency is defined as the average simulation time until a packet of the Simple Flooding algorithm reaches a node. We expect an increasing latency with growing number of nodes in the simulation, since collisions become more likely and a node has to wait longer until it receives its first packet. The latency should behave similarly in the static and mobile scenario, since the time until all nodes are flooded is only marginally higher in the mobile scenario and the nodes are expected to move only a few centimeters.

The results for the static scenario and mobile scenario are depicted in Figure 8.6 and Figure 8.7 respectively.

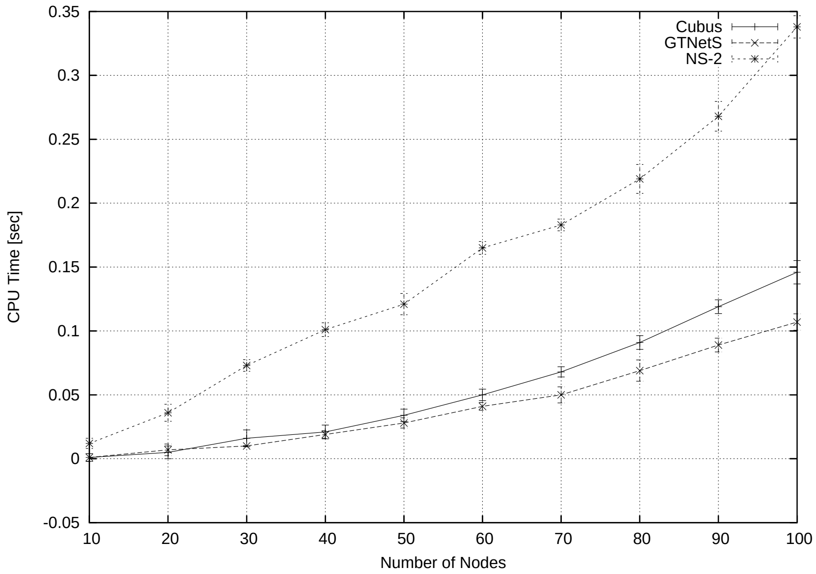

Figure 8.4: Runtime comparison with mobile scenario

For Cubus, the average simulation time in the mobile scenario is only marginally higher than in the static scenario which is an expected behavior.

The average simulation time of GTNets is higher in the static scenario. This result is unexpected and may arise from the random placement of the nodes. Like GTNetS, NS-2 shows an unexpected behavior. This behavior may also be caused by the random placement of nodes in the simulation area.

Future measurements should include a higher number of simulation runs and should ensure that the same random topology is used in all examined simulators.

8.5 Summary

In this section, we have evaluated the performance aspects and behavior of our extended Cubus system. Since, at the current stage our prototype lacks more complex environmental models and a fully flexible (graphical) user interface, we can only evaluate the mobility engine. However, since supporting mobility has been one of the most important requirements, we get an insight into the behavior of Cubus as a simulator for mobile networks. Whenever appropriate, we have compared the results with NS-2 and GTNetS.

First, we have examined the setup time of all three network simulators when nodes are mobile. GTNetS still has the lowest setup time. However, the setup time of Cubus remains linear and has not degraded, even though definitions of environmental model and mobility models were added.

Cubus outperforms GTNetS and NS-2 regarding runtime, even in the mobile scenario, when all nodes are able to move. The runtime of Cubus grows linearly. However, we have to repeat these measurements, when we support more detailed protocols and algorithms, since we then expect a degrade in the runtime behavior of Cubus. The runtime should remain in linear order.

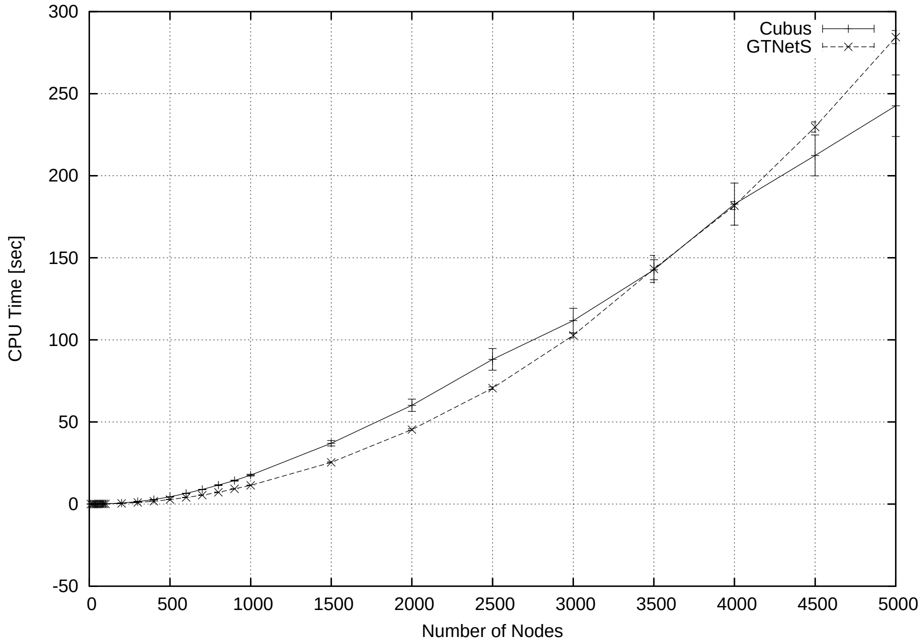

Figure 8.5: Runtime comparison with larger mobile scenarios

The latency measurements for the static and mobile scenario in Cubus have shown an expected behavior. GTNetS and NS-2 both have shown an unexpected behavior, since the latency in the mobile scenario was below of its static counterpart. We assume these results arise from the random placement of nodes in the simulation area.

This chapter has shown that the extensions to Cubus have not degraded the performance and behavior of the simulator in an unexpected way. However, we have to evaluate the system again, when more complex environmental models and a fully featured graphical user interface are developed

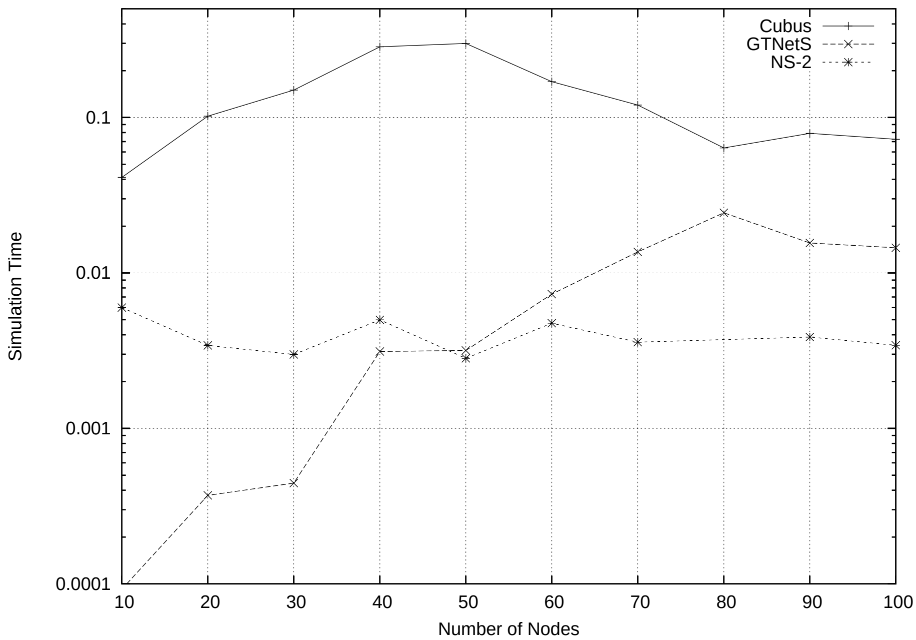

Figure 8.6: Latency in the static scenario

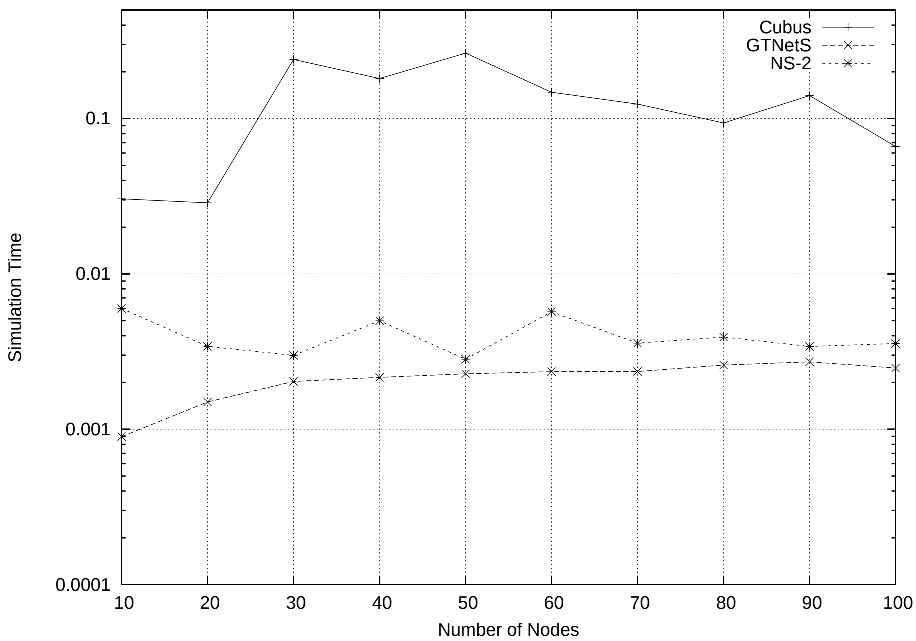

Figure 8.7: Latency in the mobile scenario

9 Summary and Outlook

9.1 Summary

Cubus is a modular, discrete event simulator which is written in C++ and was developed by [Molt04]. It supports the dynamic loading of algorithms. Due to its modularity and the dynamic loading of components during runtime, it outperforms common networks simulators, like NS-2 [FaVa03] and GTNetS [Rile03].

In this thesis, we extended the Cubus system to support the mobility of nodes and environmental aspects, e.g. propagation obstacles. Moreover, we have extended the system to support user interactivity and analysis capabilities.

In Chapter 2 we have discussed the basics of mobile ad hoc networks and have examined mobility models, since they model node movement in network simulators. Moreover, we have briefly taken a look at three propagation models which model signal propagation.

The existing Cubus system which was developed in the former thesis, is described in Chapter 3. We have listed the additional requirements which turn Cubus in a simulator not only for fixed networks but also specifically for mobile ad hoc networks. Our requirements concern the modeling of node movement, the support for spatial constraints and the support for user interactivity. We also need to support analysis capabilities for later retrieval of simulation results.

In Chapter 4 we have examined how common network simulators, like NS-2, GTNetS and OMNeT++ [Varg01], solve the tasks of node movement, spatial constraints, and user interactivity. We have concluded that GTNetS and OMNeT++ deal with mobility in the the most flexible way by providing mobility models. All three simulators do not support the modeling of spatial constraints.

When dealing with interactivity and animation, we have to consider two aspects. First of all, due to the non-linear time jumps in discrete event processing, the animation speed is seldomly constant. We have taken a look on HiSAP [Brod00], as it is the only simulator which supports real-time intervals between two event time stamps. The major problem when considering interactivity is inconsistency. We have to make sure that the user interface always has the current state of the simulation. Since it is the only simulator which supports user interactivity and its treatment of inconsistencies, OMNeT++ was examined.

In Chapter 5 we have introduced the design of our environmental aspects. We have split up the design into the *mobility engine* and the *spatial model*. The mobility engine is comprised of the *mobility models* and the *location service*. Moreover, *positioning devices* belong conceptually to this component . Mobility models define the movement of their associated nodes. All mobility models are derived from a common base class, hence they all have a common interface. The parameters of a mobility model are defined in a separate configuration file and are read in during start of the simulation. The position of a node is only recomputed, when another component requests for it.

The location service can be seen as the interface between the mobility models and the rest of the system. It forwards position queries for a specific node to the appropriate mobility model. Moreover, the location service computes all nodes which are in a geographic range of a sending node. As the retrieval of such information should be fast, the location service should be designed efficiently. We have examined the quadtree structure for a possible storage location of the nodes.

Positioning devices are attached to a node and compute a node's position by querying the location service. Geometric and symbolic positioning devices are supported in Cubus. All devices are derived from a common base class and hence have a common interface. Since positioning devices support only specific position formats, we introduce location models which map the position given in one format in the format of another coordinate system.

The *spatial model* is comprised of the *environmental model* and the *location models*. The environmental model is a container for all spatial constraints. It contains mobility areas in which nodes move, propagation obstacles which attenuate signals, GPS obstacles, and location models. Since the environmental model has a well-defined interface, the user can easily add and remove spatial constraints.

In Chapter 6, we have introduced the *interactive engine* and the *analysis engine*. The interactive engine extends Cubus into the the field of demonstrators. It contains all user interfaces. At the current state, we support a graphical user interface and a command-line user interface. The command-line user interface can be used when simulation should run as fast as possible, whereas the graphical user interface can be used for demonstration purposes.

The information about all available parameters of a component is stored in a specific *repository*, e.g. the mobility model repository stores all parameter of all available mobility models. The *parameter repository* offers a common interface to all repositories. It serves as a name service and hands parameter requests to the appropriate repository. Names are built according to an URI-like scheme, where the parameter repository forwards the name to the appropriate repository for retrieval. The parameters of an instantiated component can be retrieved and set by generic get and set methods which contain the name of the parameter as argument.

To retrieve the current state of the simulation, the *graphical user interface* synchronizes itself in intervals with the core engine. While this synchronisation takes place, the simulation pauses and no further events are processed.

The *analysis engine* is the component responsible for the processing and storage of log events. Log events can be stored in a common text file or in a relational database. Since I/O operations usually need more computation time, the analysis engine is equipped with its own event queue, in which the log events are put and which runs asynchronously to the rest of the system.

The implementation of the extensions is discussed in Chapter 7. At the current stage, we support three mobility models, the *Random Waypoint Model [BMJH+98]*, the *Graph-Based Model [SHBT+03]* and the *Static Model* in which all nodes are immobile. The environmental model stores all mobility areas and propagation obstacles in STL maps [Sili04] . Currently, a simple environmental model is implemented, using a rectangle and a graph as mobility areas.

In Chapter 8 we have evaluated our Cubus system with regards to performance and behavior. We have shown that Cubus has the lowest runtime compared to GTNetS and NS-2.

At the current stage, our Cubus system offers a flexible modular base for further development. Since it is at a very early stage, we list some of the extensions which may be realized in Cubus at a later point of time.

9.2 Outlook

The currently available version of our simulator is a solid base for simulating wired and wireless, mobile networks. Still, it lacks some improvements and extensions regarding the components in this thesis as well as other possible enhancements.

9.2.1 Mobility Engine

Mobility models define the movement of their associated nodes. However, these computations can become very complex and computationally intensive. Therefore, it could be efficient to outsource all mobility models to a more powerful machine. Then the core engine could run on an average PC, while the mobility models run on a more powerful machine that allows a faster computation of a node's position.

In Chapter 5 we have briefly discussed the different architectures for the location service to store mobile nodes efficiently. This topic could be discussed in another thesis in more detail.

9.2.2 Spatial Model

In the near future, Cubus should support more complex environmental models which contain propagation obstacles and more complex mobility areas. Propagation obstacles should not only be modeled as rectangle or line, but should resemble their real-life counterparts. For example, walls have an attenuation factor for signals which we may want to integrate in our simulator.

In the current version, we have not implemented location models. A symbolic location model would be a valuable extension to our system, since this would enable us to model realistically infrared devices, e.g. for applications in the Nexus [HKLR+99] field. Furthermore, an environmental model designer should be supported as an external tool for the Cubus system.

9.2.3 Interactive Engine

For efficiency reasons, we should parallelize the interactive engine, i.e. the graphical user interface runs on a different machine than the simulation. When running on the same machine, the graphical user interface accesses the objects of the simulation directly to set or get parameters by using their generic get and set methods. If the graphical user interface runs on a different machine, the direct access to objects is not possible, since they are located in another adress space. We have to provide stub objects, like in Corba [Obje04] or Java RMI [Sun 04]. The stub objects then forward the requested method to the appropriate object.

The URI-like naming scheme for the parameter repository is not implemented yet. In the current version, the containers are identified by the enumeration type `cubusType`. To support the naming scheme, we have to implement name resolution. This could be done similar to the DNS system [Mock83].

9.2.4 Analysis Engine

At the current stage, we provide two different locations for the storage of log events, text files and relational databases. In Chaper 6 we have defined change events which log the old and new value of a component when the user changes a parameter. When interactivity is high, these events occur very often. The user might only be interested in a specific action being logged, e.g. "create Change Log Event, when node x is set to another energy level". We propose to realize the filtering of log events in the next version of the simulator.

Additionally, we should support statistical methods and classes, e.g histogramms, which evaluate the entries in the text files or databases. They can provide useful information for the user about the performance of the simulation run.

9.2.5 Other Components

The following enhancements will also be beneficial to the Cubus system:

When simulating wireless networks, we have to model the possibility of network partitioning. This can be achieved through a *partition probability engine* that uses underlying distribution functions to model partitions.

A *load generator* should model background noise to the wireless medium by feeding additional communication overhead into the simulation scenario. It could be connected to the node runtime engine.

Bibliography

[AhSU86] A. V. Aho, R. Sethi and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley. 1986.

[Ball96] P. Ball. Introduction to Discrete Event Simulation. In *DYCOMANS workshop on Management and Control : Tools in Action*, May 1996, p. 367–376.

[BMJH+98] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu and J. Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Mobile Computing and Networking*, 1998, p. 85–97.

[Brod00] T. Brodbeck. Erweiterung des Animationssystems HiSAP um Synchronisationsmechanismen auf der Grundlage logischer Zeit. Diplomarbeit DIP-1869, Universität Stuttgart, 2000.

[CaBD02] T. Camp, J. Boleng and V. Davies. A Survey of Mobility Models for Ad Hoc Network Research. *Wireless Communications & Mobile Computing (WCMC): Special Issue on Mobile Ad-Hoc Networking* 2(5), 2002, p. 483–502.

[CaSS02] D. Cavin, Y. Sasson and A. Schiper. On the Accuracy of MANET Simulators. In *Proceedings of the Workshop on Principles of Mobile Computing (PCOM'02)*, October 2002, p. 38–43.

[CoMa99] M. Corson and J. Macker. RFC 2501 - Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations. http://www.faqs.org/rfcs/rfc2501.html, 1999.

[Davi00] V. Davies. Evaluating mobility models within an ad hoc network. Master thesis, Colorado School of Mines, 2000.

[Domn01] S. Domnitcheva. Location Modeling: State of the Art and Challenges. In *Proceedings of the Workshop on Location Modeling for Ubiquitous Computing*, Atlanta, Georgia, USA, September 2001. p. 13–19.

[Dros03] T. Drosdol. Unterstützung symbolischer Koordinaten im Lokationsmanagement. Diplomarbeit DIP-2062, Universität Stuttgart, Stuttgart, Germany, 2003.

[DüRo03] F. Dürr and K. Rothermel. On a Location Model for Fine-Grained Geocast. Seattle, Washington, USA, 2003.

[EHHM+99] D. Estrin, M. Handley, J. Heidemann, S. McCanne, Y. Xu and H. Yu. Network Visualization with the VINT Network Animator Nam. Technical Report 99-703b, University of Southern California, March 1999.

[Euro04] European Organisation for the Safety of Air Navigation. World Geodetic System 1984. http://www.wgs84.com/, 2004.

[FaVa03] K. Fall and K. Varadhan. *The ns Manual*, 2003.

[Frii46] H. T. Friis. A note on a simple transmission fomula. *Proc IRE 34*, 1946.

[GDB04] *GDB - The GNU Project Debugger*, 2004.

[GPS95] Global Positioning System Standard Positioning Service Signal Specification, June 1995. 2nd Edition.

[HKLR+99] F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel and M. Schwehm. Nexus - An Open Global Infrastructure for Spatial-Aware Applications. Technical Report TR-1999-02, 25, 1999.

[HoAk95] J. Ho and I. Akyildiz. A Mobile User Location Update and Paging Mechanism Under Delay Constrains. In *Wireless Networks*, 1995, p. 413–425.

[JBRAS05] A. P. Jardosh, E. M. Belding-Royer, K. C. Almeroth and S. Sari. Real-Worl Environmental Models For Mobile Network Evaluation. *Journal on Selected Areas in Communications special issue on Wireless Ad hoc Networks*, 2005. to appear.

[Labo04] L. L. N. Laboratory. Posix Thread Programming, 2004.

[LeMB92] J. Levine, T. Mason and D. Brown. *Lex & Yacc*. O'Reilly. 2nd, 1992.

[LoWi04] M. Loebbers and D. Willkomm. A Mobility Framework for OMNeT++ User Manual. http://mobility-fw.sourceforge.net/manual/index.html, 2004.

[MaBJ99] D. Maltz, J. Broch and D. Johnson. Experiences designing and building a multi-hop wireless ad hoc network testbed, 1999.

[Mock83] P. Mockapetris. RFC 882 - Domain names: Concepts and facilities. http://www.faqs.org/rfcs/rfc882.html, 1983.

[Molt04] M. Moltenbrey. Entwurf eines adaptiven und interaktiven Systems für die Analyse von Algorithmen zur Datenverwaltung in mobilen Ad-Hoc Netzen. Diplomarbeit DIP-2195, Universität Stuttgart, 2004.

[Mona98] C. Monarch. The CMU Monarch Project's Wireless and Mobility Extensions to NS, 1998.

[Nort00] J. Norton. Dynamic Class Loading for C++ on Linux. *Linux Journal*, May 2000.

[Obje04] Object Management Group. Common Object Request Broker Architecture, 2004.

[Open01] I. Open GIS Consortium. Open GIS Implementation Specification: Coordinate Transformation Services, 2001.

[Oual02] S. Oualline. *Practical C++ Programming*. O'Reilly & Associates. 2nd edition, 2002.

[Page91] B. Page. *Diskrete Simulation - eine Einführung mit Modula-2*. Springer Verlag. 1991.

[PrKP02] P. Prinz and U. Kirch-Prinz. *C++: Lernen und professionell anwenden*. mitp-Verlag, Bonn. 2nd edition, 2002.

[Rapp96] T. S. Rappapor. *Wireless communications, principles and practice*. Prentice Hall. 1996.

[RaTr99] P. Raines and J. Tranter. *Tcl/Tk in a Nutshell*. O'Reilly. 1999.

[Rile03] G. F. Riley. The Georgia Tech Network Simulator. In *ACM SIGCOMM Workshops*, 2003.

[Same84] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16(2), 1984, p. 187–260.

[Schi00] J. Schiller. *Mobile Communications*. Addison Wesley. 1st edition, 2000.

[SHBT+03] I. Stepanov, J. Hähner, C. Becker, J. Tian and K. Rothermel. A Meta-Model and Framework for User Mobility in Mobile Networks. In *Proceedings of 11th International Conference on Networking 2003 (ICON 2003)*, Sydney, October 2003.

[Sili04] Silicon Graphics, Inc. Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl, November 2004.

[Step02] I. Stepanov. Integrating Realistic Mobility Models in Mobile Ad Hoc Network Simulation. Diplomarbeit DIP-1989, Universität Stuttgart, Stuttgart, Germany, 2002. English.

[Sun 04] Sun Microsystems. Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi/, 2004.

[TaBa99] D. Tang and M. Baker. Analysis of a metropolitan-area wireless network. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, 1999, p. 13–23.

[THBS+02] J. Tian, J. Hähner, C. Becker, I. Stepanov and K. Rothermel. Graph-Based Mobility Model for Mobile Ad Hoc Network Simulation. In *Proceedings of the 35th Annual Simulation Symposium*, San Diego, California, USA, April 2002. p. 337–344.

[Trol04] Trolltech Inc. Qt Documentation. http://doc.trolltech.com/, 2004.

[Varg01] A. Varga. The OMNeT++- Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, June 2001.

[WHFG92] R. Want, A. Hopper, V. Falcao and J. Gibbons. The Active Badge Location System. In *ACM Transactions on Information Systems*, January 1992, p. 91–102.

[YoLN03] J. Yoon, M. Liu and B. Noble. Random Waypoint Considered Harmful. In *Proceedings of INFOCOM. IEEE, 2003*, 2003.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfaßt und nur die angegebenen Quellen benutzt zu haben.

____

(Serena Fritsch)