NS-Mapper:
Ein Szenario Editor
für Mobile Ad-hoc Netze

Emanuel Eden

March 25, 2007

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund

Diploma Thesis

NS-Mapper:
Ein Szenario Editor für Mobile Ad-hoc Netze

von
cand. inform. Emanuel Eden

Aufgabenstellung und Betreuung:
Prof. Dr.-Ing. Lars Wolf und Oliver Wellnitz

Braunschweig, den March 25, 2007

Erklärung

Ich versichere, die vorliegende Arbeit selbständig und nur unter der Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Braunschweig den March 25, 2007

Abstract

In our days mobile communication is an essential part of our environment and there is a tendency of increasing wireless independence of this technology. In case that the environment once does not support the usual infrastructure for using these techniques, spontaneous unstructured ad-hoc networks become necessary, which supply us with the important services, that allow us to communicate with each other.
Ad-hoc networks often are neither planned nor tested in real environments, they are only simulated in tests to save time, manpower and money. While the simulation of wireless networks is already well developed, the random modelling of nodes is still based on very simple strategies, which have to be improved.
In this thesis an editor for scenarios of mobile ad-hoc networks, the NS-Mapper, is improved and extended by adding more realistic strategies, such as random based node placement, movement and traffic. Preexisting algorithms were collected and inspected for these tasks. For every task a more realistic random algorithm is chosen and integrated into the Software. Furthermore, a flexible plugin structure, within the NS-Mapper was developed for a smooth integration of the random part.
Finally the developed realistic random based algorithms are compared with the available simple strategies - for example the random mobility model - and it is described, what the differences between these strategies are, and if they are of higher quality and quantity compared to the simulated arrangement.

Kurzfassung

Mobile Ad-hoc Netze sind spontan entstehende Funknetze, bei denen mobile Geräte miteinander Daten austauschen. Um die recht kleine Funkreichweite des Senders zu vergrößern, leiten die Teilnehmer des Netzes Daten durch das Ad-hoc Netz bis zum Ziel weiter. Aufgrund der Größe von Ad-hoc Netzen und der Anzahl an mobilen Geräten werden Untersuchungen in diesen Netzen meist nur simuliert. Während die Modellierung der Datenkommunikation in mobilen Netzen dabei recht gut erforscht ist, beschränkt sich die Simulation der Bewegungen von mobilen Geräten meist nur auf sehr einfache Modelle.

In dieser Diplomarbeit soll ein Editor für Szenarien mobiler Ad-hoc Netze, der im Rahmen einer Studienarbeit entstand, weiterentwickelt werden. Der Schwerpunkt dieser Arbeit liegt dabei auf der Entwicklung von Zufallsstrategien für die Plazierung, die Bewegung und den Datenverkehr von mobilen Geräten. Nach einer Untersuchung verschiedener existierender Ansätze für die o.g. Strategien, soll jeweils ein Ansatz jeder Klasse implementiert werden. Die Strategien jeder Klasse sollen dabei austauschbar und modular ausgelegt werden, die Schnittstelle zum Editor ist dabei entsprechend zu definieren und ausführlich zu beschreiben. Ein abschließender Vergleich der hier entwickelten Lösungen mit den üblichen Ansätzen zur Plazierung und Bewegung von mobilen Knoten (Random placement, Random-Waypoint Mobility Model) soll die Unterschiede qualitativ wie auch quantitativ beleuchten.

Task Description

Contents

List of Figures

Listings

Chapter 1

Preface

Tests are often made in simulations because it is not always practical to imitate or repeat situations in a real environment. Simulations are the reproduction of real processes and systems, using simplified models. In this diploma thesis, we deal with the simulation of ad-hoc networks.
Simulations of ad-hoc networks are often used by scientists to save resources, because it is too delicate and time-consuming to plan scenarios for protocols and test them directly in a real situation, only to discover that they is not working properly. Therefore, if the scientists do not want to waste time and energy, it is necessary to make suitable software for a user-friendly simulation of these tasks commonly available. It is not efficient if scientists always start their work from zero and plan scenarios only for their own use. That is the reason why the Network-Simulator 2 (NS-2) was developed and in this class additionally the NS-Mapper, which is a scenario editor for ad-hoc networks.

1.1 NS-2 and NS-Mapper

The NS-2 is one of the most used simulators for network research, which provides for example: routing, multicast, TCP and UDP support for wired and wireless networks. While the simulation is already comprehensive and stable, the modelling and the analysis of data is only limited to many small specialized tools, which are additionally not very user-friendly and partially not well documented. Many developers, who are not familiar with the NS-2, often do not find the proper tools for their problems and thus, are forced to generate their own models and utilities.
The development of wireless ad-hoc network simulations can grow fast to

complex structures. For these complex structures a modeller for wireless networks is missing, which combines many possibilities of the NS-2. The user is forced to generate his simulations step by step, if he wants to have parts of the scenario, which then, for example, have only limited movement behaviours. This is a waste of time and manpower, because the realisation of scenarios often has a common denominator and the settings only have to be adjusted to these cases.
To improve this situation, the NS-Mapper was developed to model wireless networks in a prior study thesis. It offers an easy way to plan and model various scenarios and to validate their attributes. The software provides common modelling functions, like copy, cut and displace functions and in its present state, it also contains simple random node- placement, movement and traffic algorithm functions (lookup 1.2 and 3).
The basic elements to control the node movement behaviour in scenarios are the MovementFields, which limit the node movements to closed areas. The MovementFields can move within the entire map, whereas the wireless nodes can only move within the MovementFields. By this function it is possible to simulate apartments or whole environments with streets and vehicles in 2D.
Even if the NS-Mapper was primarily developed to assist the NS-2, this modeller can also supply any other network simulator by the development of new export plugins.

1.2 Focus of this Thesis

Wireless radio networks, which appear spontaneously, are called ad-hoc networks. In ad-hoc networks the wireless equipments profit from the routing between each other. If they do not have direct access to a base-station or if the devices want to communicate with a remote host in the same ad-hoc network, they themselves can connect to each other and thus, spread their working radius enormously. But ad-hoc networks can grow too big, and then these complex figures and compositions have to be analysed, preferably in an easy and cheap way. For this reason the ad-hoc networks can be analysed in simulations, which examine the efficiency of a routing protocol. In contrast to the modelling of data communications between mobile equipment, which are well explored, the simulations of movement models are only done

in simple courses of events.
As described above, the NS-Mapper is a modeller for ad-hoc networks, which, in this thesis, is extended by random strategies. These extensions contain, at first, the random start point functions, which place the wireless nodes into the scenario in a naturalistic way. Secondly, a random algorithm of realistic movement behaviour is chosen. And thirdly, a random traffic algorithm is generated, which produces an objective data flow in a model. For these three random functions pre-existing algorithms are collected, examined and finally chosen for the implementation. In case a random algorithm for a task already exists, it is taken, if not, an own realistic strategy is developed. The random algorithms are integrated smoothly in the NS-Mapper by using a plugin structure. These plugin structures encapsulate different programme parts from the core program. These structures are designed in a modular way, which allows the different parts of the implementation to be exchanged and reused in other plugins.
Finally, the implemented random strategies have to prove their capabilities in a proper simulation. This will, on one hand, show their functional efficiency and on the other hand, demonstrate the differences to usually used algorithms, for example the Random-Waypoint mobility model.

1.3 Organisation of this Thesis

In the chapter 2 (Fundamentals) most candidates for random algorithms of the previously mentioned tasks: random start point, movement and traffic, are introduced. There advantages and disadvantages are discussed and finally, it is decided, which of them have the most realistic character and therefore, are integrated in the NS-Mapper. Furthermore this section deals with other network modellers, scenario editors and the algorithms they use.
In chapter 3 (DesignConcepts), the plugin structure is planned and the different ways, how to realize the course of events, are discussed. Additionally, the requirements of the data structure, regarding how a plugin receives its needed data and how to process plugin functions without compromising the stability of the core programme, are explained. Then it is described in detail, which classes are relevant for additional functionality and comfortable plugin development, for example how to exchange data between the different plugins, how to access the painting area for presenting extra information or

how to catch mouse events for influencing random functions.
Chapter 4 (Implementation) explains the implementation of the selected random algorithms and their realizations. It describes the design concepts of plugins and their framework. The different processes of establishing these structures and classes are explained and their advantages and disadvantages are discussed. Additionally, this chapter addresses security aspects, which protects the processes and structures of the core part of the programme. Furthermore, it deals with the important aspect of the reusability of the structures of this design concept.
In chapter 5 (Evaluation) we describe the planning, the execution and the analysis of test scenarios, which allow a direct comparison between the newly (chapter 4) developed random plugins and the already implemented plugins of the former study thesis. The random plugins will be evaluated regarding their effectiveness.
The last chapter 6 (Conclusions and Outlook) provides a conclusion about the procedures and functionality of the software and gives an outlook of the direction, in which the NS-Mapper can further developed.

1.4 Acknowledgements

Ich moechte hiermit meinen Dank und meine Verbundenheit an die Menschen richten, die mich bei meiner Diplomarbeit unterstuetzt haben. Besonderer Dank gilt meiner Familie, die mich immer in allen Belangen und selbst in tiefer Aussichtslosigkeit, nicht nur geliebt und unterstuetzt, sondern sich aufgeopfert haben. Ich moechte zudem Oliver Wellnitz danken, dass er mit mir mein Interesse, meine Freude und mein Hobby geteilt hat. Desweiteren moechte ich Herrn Professor Dr. Lars Wolf danken, der mir ermoeglichte, meine Arbeit am NS-Mapper weiterzufuehren.

Chapter 2

Fundamentals

This chapter begins with an overview of the different tools, which are used for random node placement, movement and traffic generation to give a presentation of the presently available possibilities to automate ad-hoc simulations.
Then we enumerate and discuss the different methods and algorithms of the random ad-hoc generation. The different categories are examined in regard of their realistic and natural behaviour, although it is not always easy to differentiate between unrealistic, realistic and natural behaviour, because the algorithms have not been verified in evidence studies. To give an example for the difficulties to differentiate between realistic and natural behaviour: to move forward and to go left or right is more probable than backwards. This is realistic behaviour, but this does not guarantee an image of natural behaviour at the end. However, at the end of this chapter, we have to come to a decision of what is integrated in the NS-Mapper.

2.1 Existing ad-hoc scenario editors and traffic generator

Even if the NS-Mapper is developed as an open ad-hoc modeller and therefore, can be used in many different network simulators, the initial intention of its creation was to integrate missing modelling functions in the NS-2. In this section, particularly those modellers and extension tools are listed and described, which are able to operate in the NS-2 system.

The following modellers and simulators of Ad-hoc networks exist:

ANSim

ANSim[13] is a simple Ad-Hoc network simulator. It examines connectivity and is suited for statistical simulation with long simulation times and large networks. The simulator has two kinds of nodes: static nodes, which are only placed and then remain at this position, and mobile nodes. The mobile nodes can be placed in a uniform distributed way, by a Gaussian and an elliptical distribution. The supported mobility models are Random Direction, Random WayPoint and Gauss Markov. All nodes are generated automatically. If the user prefers to generate his own mobile nodes, he has to import them by using a configuration file. The calculations are presented graphically in the simulator or in trace files. Its documentation does not give any information about the used protocols.

GloMoSim

GloMoSim[26] is a scalable network simulator for cable and wireless networks. It has various routing protocols, for example AODV and DSR, and supports TCP and UDP as transport protocols and applications, such as CBR, FTP, HTTP and Telnet. It also contains some modelling functions, such as a uniform distributed placement for random start points of nodes, and it supports some mobility models like Random Waypoint, Random Drunken Walk and Trace based mobility.

2.1.1 Ad-Hoc modeller

CanuMobiSim

The CANU Mobility Simulation Environment or CanuMobiSim[19] is a framework for mobility modelling. The Framework is implemented as a Java application and can be used for the following simulators and emulators: NS-2, GloMoSim and QualNet. The CanuMobiSim uses a XML based scripting language to define the simulation environment. It is not a GUI based application. It can calculate mobility models, such as: Random Walk (Brownian Motion), Gauss Markov, Random WayPoint, Graph based mobility model, as well as micro mobility models, such as intelligent driver motion, smooth motion and constant speed motion.

USC mobility generator tool

The USC[23] is a mobility scenario generator framework, which consists of a set of different tools, that can calculate the Random Waypoint model, Freeway mobility model, Manhattan mobility model and the Reference Point Group mobility model. The generated trace files are compatible with the NS-2 and can be directly used in the NS-2 or analysed by another programme. This tool is made for a console based generation and does not have a GUI.

BonnMotion

BonnMotion[3] is a Java based application, which creates and analyses mobility scenarios. The environments can be applied to the NS-2 and the GlomoSim/QualNet simulator. It supports the Random WayPoint model, the Gauss Markov model, the Manhattan Grid model and the Reference Point Group Mobility model. The scenarios are generated by a console based programme.

2.1.2 Traffic Generator

PackMime

The PackMime[14] traffic model includes a model for HTTP traffic. The traffic intensity is controlled by rate parameters, representing the average number of new connections, started each second. Its implementation in the NS-2 is capable of generating HTTP 1.0 and HTTP 1.1 connections.

Video traffic generator

The Video traffic generator[24] is based on the Transform Expand Sample model of MPEG4 trace files. It generates traffic with the same first and second order statistics as an original MPEG4 trace.

Web traffic generator

The Web traffic generator[22] is a module for generating background Web-like traffic. A more comprehensive model that extends this basic traffic generator has been added to the NS-2, but this has not yet been documented.

BonnTraffic

BonnTraffic[18] is a stand-alone application for generating synthetic traffic for network simulations. It is a modular framework, which integrates models for a wide range of application-level protocols. The traffic can be exported to be used for the NS-2. Several traffic models are integrated, such as TCP, UDP traffic and traffic modules, such as HTTP client-server structures.

2.2 NS-Mapper

In contrast to many other tools, which generate node motion and traffic, the NS-Mapper is not limited to only one special category. The NS-Mapper connects modelling, random node placement, random ad-hoc routing and random traffic tools in only one single application, whereas most other available tools, described in the proceeding sections, can only calculate and vary scenarios as a whole at once. In other programs a direct influence or a separation of different behaviours for different nodes is not possible or at least very difficult.
The NS-Mapper editor consists of two layers: The default GUI-based version, in which scenarios can be modelled by the user and a command line version for creating simulation files from scenario descriptions automatically. A NS-Mapper scenario contains manually modelled nodes and node mobility, as well as randomly created nodes and their movement. A user can manually define the number, the position and the mobility of nodes, he is particularly interested in, as well as their traffic pattern. Additionally, he can let the editor randomly create a given number of additional nodes to form a larger ad-hoc network. NS-Mapper scenarios can then be used to create a number of NS-2 simulation files, in which all random elements are recalculated by different seed settings. This allows the creation of scenarios with a group of fixed nodes, which behave in a predetermined way, and some random nodes, which create background traffic and provide connectivity in the ad-hoc network, according to some general settings.
Another feature of the editor are the so called MovementFields. A MovementField is a rectangle or a polygon, in which mobile nodes can be created and move. Mobile nodes are not allowed to move outside a MovementField. This feature can be used to create walls or other borders, which the nodes cannot pass. A node can only move from its own to another MovementField,

when these are connected or overlap each other.

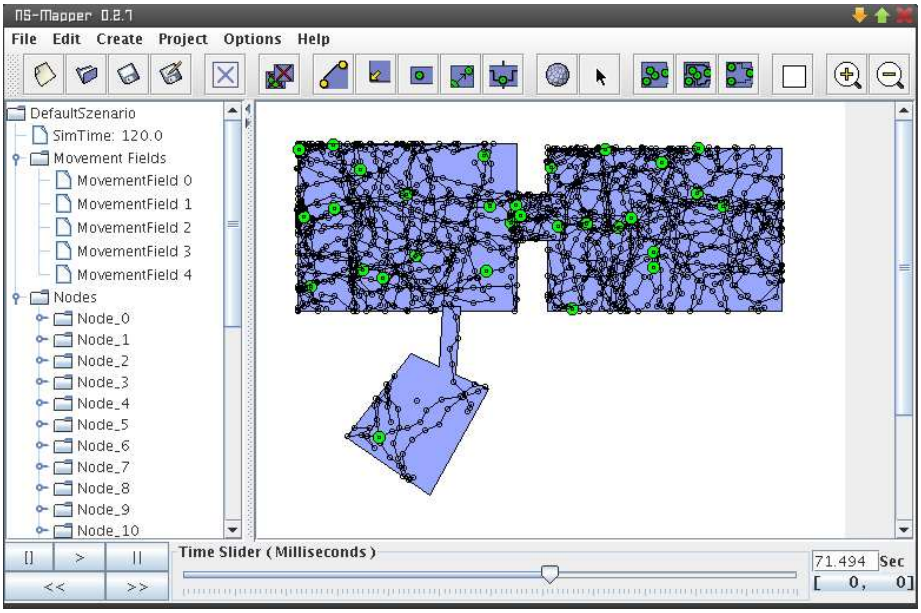

Figure 2.1: The NS-Mapper scenario editor

Figure 2.1 shows a screenshot of the NS-Mapper GUI with some static MovementFields (blue with black borders) and some placed nodes (green). The black lines in the simulation picture represent their movement, which is generated by the Gauss Markov algorithm.
There are two kinds of MovementFields: static and dynamic fields. Static MovementFields remain at their position during the simulation. For dynamic fields, the user can create one or more time-based movement vectors, which contain start time, stop time, speed and/or destination. All mobile nodes move with their associated MovementField which means that a movement vector of the field is added to the movement vector of its nodes. Basically, dynamic MovementFields move around with the mobile nodes sitting or moving on top of them. Examples of a static MovementField are paths, were nodes can move along on pavements, streets, or rooms in buildings. Dynamic MovementFields can be used to create moving objects, such as busses, trains or other vehicles or transportation mechanisms.
While static nodes are directly created by the user, the NS-Mapper can also create an arbitrary number of random nodes. Random nodes are placed

according to a given random node placement strategy. Another strategy concerns random node movement. Any node, independently whether it was placed by the user or created randomly, can be chosen to move according to a random node movement strategy. For static nodes, this allows random movement with a fixed start point in all simulations of a scenario.
Apart from the setup and the movement of mobile nodes, also data traffic has to be modelled for a scenario. The user can create arbitrary CBR and/or several different TCP traffic patterns between static and/or random nodes. Furthermore, additional traffic can also be generated automatically by NS-Mapper.

2.3 Existing randomised Algorithms

In this section, we will present most of the existing random algorithms for random start point, random motion and random traffic in detail.

2.3.1 Node placement

Although a realistic node placement is very important for random based ad-hoc modelling, a fundamental evaluation or a tool for this evaluation does not exist. In general, the modellers use a uniform distributed random calculation, even if the simulations are based on behaviour patterns, like the group motion of nodes. Therefore, here we enumerate some ideas, which can be used for the NS-Mapper.

Uniform Distributed Node Placement

The most commonly used algorithm for node placement in a simulation is the "Uniform Distributed Node Placement" strategy, which puts the nodes all over the scenario. Thereby, the nodes are placed, by assigning them a value on the x and y axis by a uniform distributed random value technique. The researchers know about the problems of this placement technique, because it ignores that people like to form groups. Nevertheless, it is an often used strategy because it is easy to implement and additionally, the number of alternative strategies of random algorithms for node placements is very limited. Furthermore, no public evalutation study has been performed, which emphasizes its disadvantages.

Gaussian Distributed Node Placement

This node placement strategy is not very different from the "Uniform Distributed Node Placement", but it tries to solve the problem of the grouping people. It uses Gaussian random numbers and places all nodes near to each other, mostly into the middle of the scenario. Up to now, it has not been evaluated if this technique is a better way to place nodes.

Mobility Motion Based Pre-Calculation

In most cases, when a random scenario takes place, the node distribution changes automatically from a uniform to a more none-uniform state, depending on the used mobility model. That is the reason why users frequently try to perform a pre-calculation for the scenarios for a limited time period. In a pre-calculation, the random nodes are distributed by an arbitrary node placement technique and then, new positions are pre-calculated by a mobility motion model. The end position of the nodes after this pre-calculation form their new start points for the simulation. It is obvious that the start point distribution depends on the used mobility motion model.
However, a pre-calculation can be circumstantial in some cases. First, if the user models a hybrid scenario with self-defined and random nodes, he has to consider many details, if he wants to change the course of events, such as for instance, the time of the pre-simulation. Secondly, the modeller has to decide how the self-defined nodes react in the time of the pre-simulation, for example, if they move or if they remain at their position. Thirdly, he has to exclude the pre-simulation from the following analysis.
A "Mobility Motion Based Pre-Calculation" can help the user, because it automatically performs a pre-calculation of the nodes movement before assigning them their start position. That way, the pre-calculation does not interfere with the real simulation or the exported trace files. Small changes in the time intervals do not become a problem anymore, and the simulation can combine different Mobility Model Based Pre-Calculations with different mobility models, which enables a larger variability of techniques with different probability distributions.

Group Based Probability Distribution

If the motion of mobility nodes has behaviour patterns like a group formation without an arbitrary capturing, the nodes can hardly be generated by a start point technique of a uniform distribution or a motion based pre-calculation. Then, it is important to place these nodes close to each other, for example, in a defined circle or an ellipse around the centre of a virtual area or a domination node. This way, the mobility nodes have to be distributed with a minimum and maximum distance and a random direction vector around a specific point or node.

2.3.2 Node movement

As shown in the figure 2.2, in the box "Global Motion", the single mobility models are divided into two main categories. The first category is "Target Orientated", where the individual nodes gets their random targets one after the other, or it creates a list of different targets, which are defined at the beginning of the simulation. This category is divided into the "Graph Based" and the "Geometrical Based" mobility models. The Graph Based mobility models can use the algorithms and data structure of the graph theory; an example of this category is the City Section model. The Geometrical Based mobility models can use the methods of the geometry, like an intersection-calculation in limited environments. An example of this category is the Random Walk mobility model.

The other main category is composed of those mobility models, which do not define a target and therefore, create a diffuse movement. This is called "Diffuse Behaviour". Some of these mobility models possess a "Uniform Distributed" probability for all directions, such as the Random Direction mobility model. The others belong to the "Non Uniform Distributed" direction mobility models, which have directions with a higher probability to be chosen. They preferably choose those directions, which lie at a 90 degree angle of their last direction. These have two subcategories: The Graph Based model can use the same methods as the target oriented category. A substitute is the Freeway mobility model. The other subcategory contains the "Pixel Based" mobility models, such as the Gauss Markov. They control their direction after every single step and can change or correct it, if needed. For example, if the nodes come close to the environmental borders, they turn around smoothly.

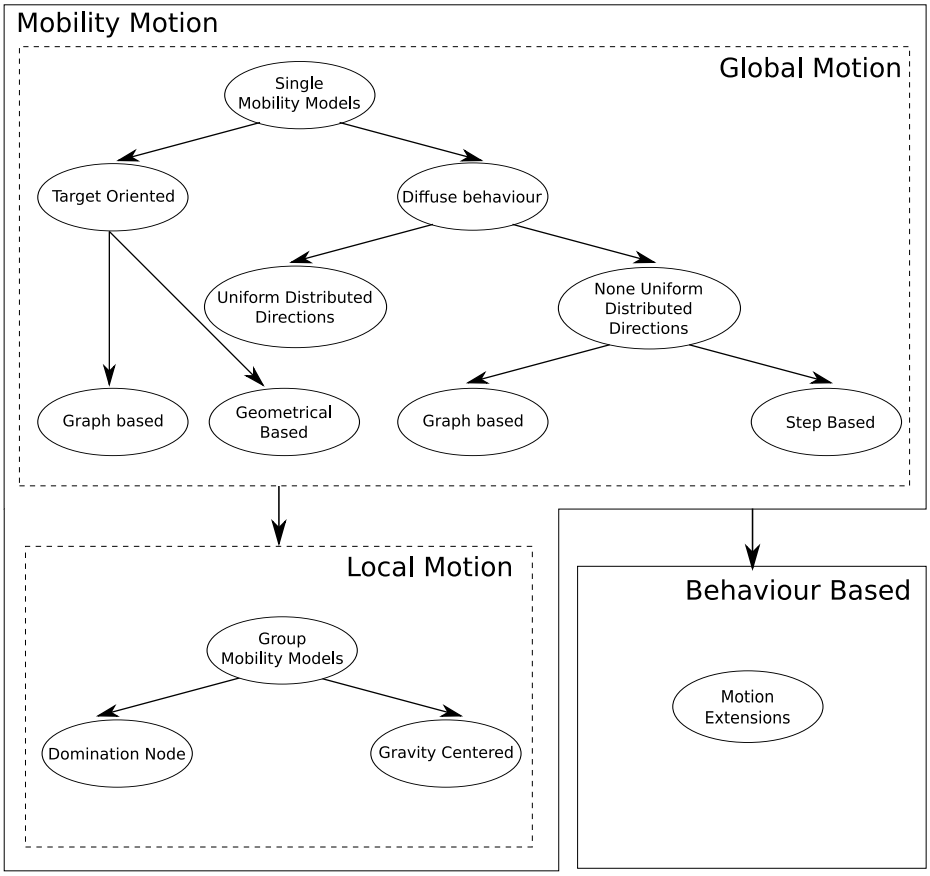

Figure 2.2: Mobility Model Categorisation

The above described mobility motion categories move in the whole scenario without any restrictions, so long as they are not locked up in areas, for example a room with walls, etc. That is why these mobility motions are referred to as "Global Motion". The group based mobility models are called "Local Motion" models, because their movement is locally restricted in depending on the maximum distance to their moving centre, which uses a global motion mobility model.

The "Local Motion" or group motion box is divided in two subparts: At first, the "Domination Nodes", which includes one node, which dominates all the others in its group. The other nodes of the group are following it or hiking around it as their centre. The "Gravity Centred" model has a virtual centre, in which the grouped nodes are captured.

The "Behaviour Based" box adds "Motion Extensions" to the mobility mod-

els. This can increase the naturalness of some models. An example of this category is the Smooth Extension, which smoothes the edges and changes the spiky direction changes of the Random Walk into a curve.

Mobility Models

Some of the mobility models make use of special functions or extensions, which first should be described.

- The 4-way and 8-way stepping is an often used method for colour flooding in computer graphics. It colorizes pixels by "stepping" in all directions till a border is reached. It also performs shortest path calculations for finding paths through a labyrinth or to wire a board, while all paths are memorized. This technique is also used for motions in some mobility models. It works on a pixel level, which limits the directions from an initial position to four or even eight independent directions, as shown in figure 2.3. A disadvantage of this method is the computational complexity and its high memory consumption, when all paths are memorised (shortest path). Unfortunately, the small number of available directions generates only a limited selection of notable changes, depending on the initial position.

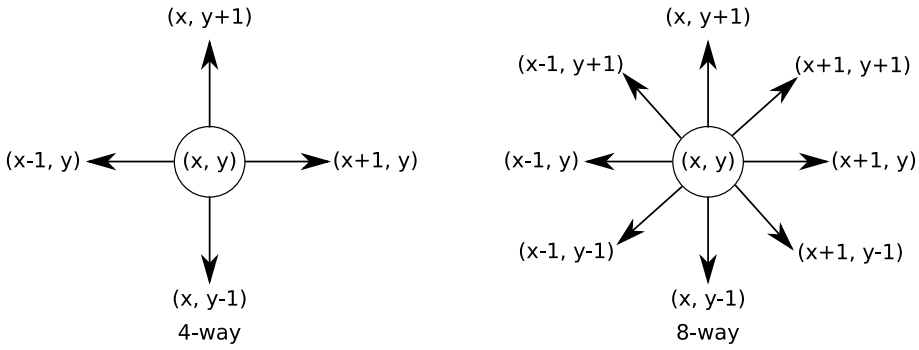

Figure 2.3: 4-way and 8-way stepping

- The Smooth Motion method is often used to generate a more realistic behaviour in "Geometrical Based" mobility models (lookup chapter 2.3.2). It smoothes spiky directional changes without calculating whole bezier curves. It is based on the recursive algorithm of De-Casteljau for non-contiguous functions, which have a small degree.

- A more natural extension, particularly for a smooth motion, is the "Driver Motion". This method is used to de- or accelerate the speed in curves in dependence on the angle of the directional change[1].

Random Walk

The Random Walk mobility model[4] is based on the idea of taking successive steps to different random directions with different velocities. It has to simulate erratic and non-predictable movements, as they often appear in nature. That is the reason why it is also called Brownian Motion or Random Drunken Walk. Every direction has the same probability to be chosen, defined by a vector with the interval [0, 2π] or by independent directions with 4-way or 8-way stepping (lookup figure 2.3). A change of direction occurs after a given time period, a defined distance or if the node reaches an environmental boundary.
Its easy implementation and its "memoryless" mobility patterns (paths are not memorised) is responsible for its common use. But this algorithm does not possess a very realistic behaviour within the environment. For images within nature, the Random Walk may be a choice, if - for example - someone wants to simulate the motion of animals. But the behaviour of a human person under normal circumstances is not well covered by this algorithm, because the directional probability of a technocratic movement of a person is not distributed uniformly. Most people walk target-orientated. Walking forward is more probable than walking backwards, so the Random Walk behaviour, created by this algorithm, appears erratic.

Random Waypoint

The Random WayPoint algorithm[15] is the most commonly used mobility model for simulations. In its basic form, the mobile nodes start uniformly distributed over the scenario. For their motion, every node gets a random target and tries to arrive at this target with a random speed. When the node has completed its movement, it waits for a given time period and then starts again with a different target.
Although the nodes are distributed uniformly, the simulation environment contains areas, which are struck more than others[28]. Figure 2.4 shows a simulation by the Random WayPoint mobility model of 5 nodes in 500 seconds with a pause time of 5 seconds. This figure shows in the second graph

(on the right) that the average node motion is focused in the centre of the simulation. But its appearance (first graph on the left) has a quite different distribution. This unbalanced motion appears relatively natural, because also in nature people cross a huge place more often and this behaviour automatically increases the probability of passing the centred area.

But the Random Waypoint Model has also some weaknesses. Even if

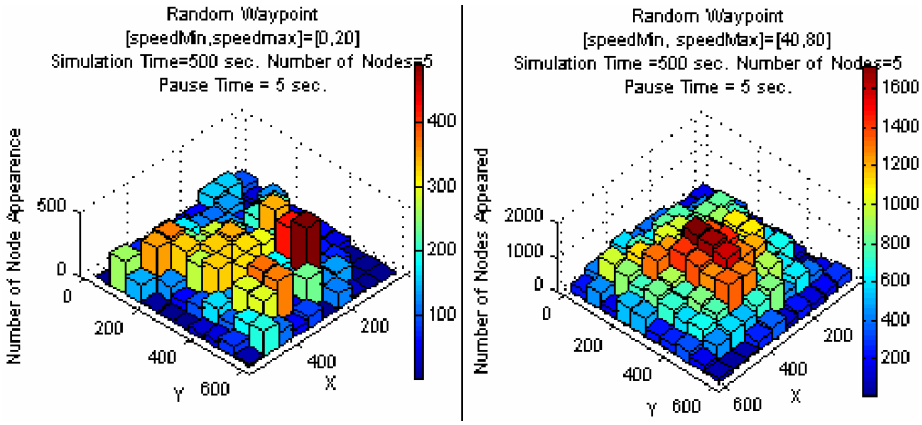

Figure 2.4: Node Distribution with Random Waypoint Mobility Model

the algorithm is target-oriented, the global effect of the movement seems chaotic, because the random targets are chosen by chance and therefore, appear unstructured. But in reality the movements of a person towards a target are considered, and they possess an individual, but still appropriate kind of order.

This cognition brings us to the next point. As shown in figure 2.4, the motion is focused in the centre of the simulation, but actually the node movements also have to be expanded into the periphery, if the algorithm wants to seem natural. For example, when people go shopping, they do not only cross the centred areas, they also walk along the borders, because there are the shops.

A further problem occurs if the start positions of wireless nodes are once randomly distributed in a uniform way, which has not a transient nature of the simulation at the beginning. To avoid this problem, the analysis sometimes starts after a short calculation warm-up period, until the nodes are better arranged.

Another difficulty are the acute angles, that the Random Waypoint produces, when the movement changes direction. A good solution to avoid this

problem is to use a method like the Smooth-Motion (lookup chapter 2.3.2), which converts the spiky direction changes of the Random Waypoint to a smooth curve.
In spite of these weaknesses, the advantage of this Mobility Model is its simplicity. It can easily be extended by a Shortest-Path algorithm like the Dijkstra or the Floyd's algorithm, if the models are more complex, for example adding areas, which can not be accessed by nodes.

Random Direction

The Random Direction Mobility Model[16] is very similar to the Random Walk Mobility Model. It was developed as an alternative to the Random WayPoint, which has a non-uniform distribution of node strides in the centre of the simulation area (lookup section 2.3.2). The difference between the Random Walk and Random Direction consists of a slight behavioural change: If a new random direction is found, the node goes straight, until the environmental border is reached, but it never changes its direction after a defined time period or after a given distance.
This behaviour pattern guarantees a more uniform distribution than the Random Walk and in contrast to that algorithm, it decreases the probability that a node moves in an unlimited area for a long time.

Boundless Simulation Area

The Boundless Simulation Area Mobility Model[12][27] belongs to the step-by-step or pixel-based mobility models, such as the "Gauss Markov" or the "Probabilistic Random Walk". The calculated steps depend on their last direction, their distance to the previous position and their current speed, whereas the speed can depend on a fixed value or a Gaussian distribution.
Just as the name says, the Boundless Simulation Area mobility model does not have any real bounds. When a node reaches one end of the environment, it continues travelling and reappears at the opposite side of the simulation area. This technique allows an unobstructed motion, like moving on a torus.
In this mobility model a relationship exists between the current direction, speed and the previous direction and its speed. In detail, we have a velocity vector $\upsilon = (\upsilon, \Theta)$, which describes a mobile nodes velocity υ and its direction Θ, while its position is represented as (x, y). Both the velocity vector and the position are updated at every $\Delta\Theta$ time step, according to the following

formulas:

$$\Theta(t + \Delta t) = \Theta(t) + \Delta\Theta$$

$$\upsilon(t + \Delta t) = min(max(\upsilon(t) + \Delta\upsilon, 0), V_{MAX})$$

for the random steps and for the positions x and y, with:

$$x(t + \Delta t) = x(t) + v(t) * cos\Theta(t)$$

$$y(t + \Delta t) = y(t) + v(t) * sin\Theta(t)$$

where V_{MAX} is the maximum velocity for the mobile node in the simulation, $\Delta\upsilon$ is the change of velocity, which is uniformly or Gaussian distributed between $[A_{MAX} * \Delta t, A_{MAX} * \Delta t]$. A_{MAX} is the maximum acceleration or deceleration of a given mobile node, $\Delta\Theta$ is the change of direction, which is uniformly distributed between $[\alpha * \Delta t, \alpha * \Delta t]$, and α is the maximum angular change of the direction of a travelling mobile node[4][5].
In contrast to the previous algorithms, the Boundless Simulation Area has a rudimentary memory. That means that forward directions are more probable than backward directions (an angle more than 180 degree). This behaviour seems much more natural. The calculations in boundless environments are strictly defined, but many simulations, for example, models with obstacles, do not tolerate this restricted behaviour, as complete directional changes are not intended for this algorithm.

Gauss Markov

Like the Boundless Simulation Area, also the Markov Gauss Mobility Model [11][4][5] is provided with a rudimentary memory of its previous moves, which guarantees a high probability of a non-uniform direction.
Initially, every mobile node has received its own direction d_n and speed s_s, as $n \rightarrow \infty$. Distance and speed are controlled by one turning parameter α, which controls the linearity of the below formula, where $0 \leq \alpha \leq 1$. Totally random values are obtained by setting $\alpha = 0$ and linear motion is obtained by setting $\alpha = 1$. The values s_{n-1} and d_{n-1} are the speed and the direction of the previous step, and s and d are constant values for the speed and the direction, representing the mean values. At last we have Gaussian random numbers with $s_{x_{n-1}}$ and $d_{x_{n-1}}$ in the following formula:

$$s_n = \alpha s_{n-1} + (1 - \alpha)s + \sqrt{(1 - \alpha^2)}s_{x_{n-1}}$$

$$d_n = \alpha d_{n-1} + (1-\alpha)d + \sqrt{(1-\alpha^2)}d_{x_{n-1}}$$

After fixed time intervals, s_n and d_n are updated.
The calculations for the Positions x and y will be done in radiant and by the following functions:

$$x = x_{n-1} + s_n * cos(d_n)$$

$$y = y_{n-1} + s_n * sin(d_n)$$

The Gauss Markov algorithm prevents the nodes from hard turnarounds, by removing them from the environmental limit without reintegrating them at the opposite side, when they reach the environmental borders, as the Boundless Simulation Area does. A further benefit is that a node is not allowed to remain near an edge for a long period.
To slow down the nodes near the borders, the whole environment has buffer zones. When a node reaches a buffer zone, the direction factor d is modulated with the angles shown in figure 2.5.

The buffer zones are an adequate way for simple scenarios, but complex

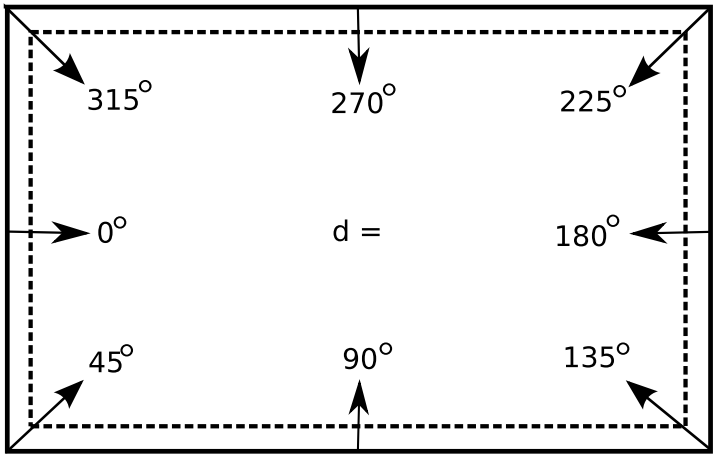

Figure 2.5: Buffer Zone of the Gauss Markov Mobility Model Environment

environments with limited node motion, caused by walls, for example, or the ground plan of a house or a whole map, can raise some problems. These conditions can provoke bottle necks or even generate dead ends (lookup figure 2.6). Especially if polygon scaling is used to generate the buffer zones, the edge calculation can loose contact, which generates separated islands in the scenario.

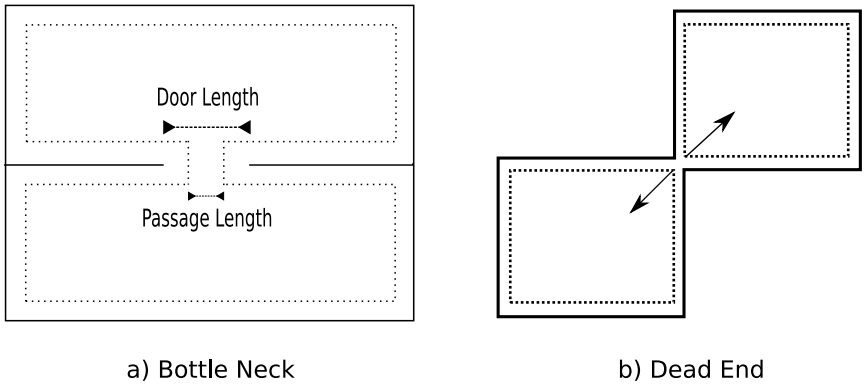

Figure 2.6: Bottle neck and dead end buffer zones

Probabilistic Random Walk

The Probabilistic Random Walk[2] mobility model uses the 8-way stepping method to generate a probable behaviour of the mobility nodes. The directional changes are made after fixed time intervals, and the different directions are represented by a matrix of directional probabilities, whereat $P(x, y)$ represents the position:

$$P = \begin{bmatrix} P(0,0) & P(0,1) & P(0,2) \\ P(1,0) & P(1,1) & P(1,2) \\ P(1,0) & P(1,1) & P(1,2) \end{bmatrix}$$

The X and the Y coordinates have three different states for their position. Figure 2.7 shows a possible distribution of probability for the different directions. State 0 does not change its position ($X' = X | Y' = Y$). In many simulations the probability for this state is zero, but that does not mean that this state never appears. It may appear in directional changes of only one step. If the probability is > 0, a horizontal or vertical direction or a waiting position is possible for the mobile node. State 1 decreases its x or y coordinate ($X' = X - 1 | Y' = Y - 1$) and state 2 increases the x or y coordinate ($X' = X + 1 | Y' = Y + 1$).

The probabilistic Random Walk is more natural than the normal Random Walk, because it generates a semi-constant forward direction. But it is very difficult to get a realistic distribution of probabilities for the matrix. And it is hard to find an evidence study, which fits for many cases, if not impossible.

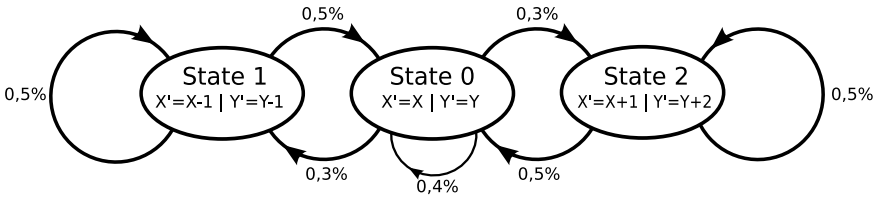

Figure 2.7: State changes of the Probabilistic Random Walk

Freeway

This mobility model imitates the motion behaviour of mobile nodes on a freeway[27], which possess a constant movement flow. The nodes do not leave their track, unless they come to a junction, in which two different directions cross each other. Only in this case, the random possibilities are calculated, and the mobility nodes can switch to another direction. The mobility nodes in a normal freeway environment move under constant conditions. Every way has a defined 4-way direction and a fixed speed.

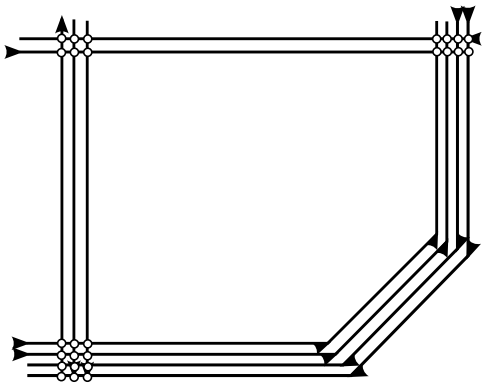

Figure 2.8: Freeway

Some Freeway derivates have a random speed, with natural acceleration and well-defined distances between the nodes, because the passing manoeuvre is not allowed in this concept. The slower nodes impair the motion of faster nodes, which can be very realistic in special situations. But sometimes it is not, because the mobile nodes can only change their movement behaviour at defined junctions. This behaviour can be harmful and quite unrealistic, if, for example, one lane has a traffic jam and its outside lane with the same direction is totally free.

Manhatten

The Manhatten[27] mobility model is one of the above mentioned derivates of the Freeway model. But in contrast to the Freeway model, the Manhattan has a regular web of streets, each consisting of two opposite directional lanes. If the nodes come to a junction, the random direction is limited to three ways. The distribution of probability is 50% for the straight direction and 25% each for the left and the right direction.

City Section

The City Section mobility model[21] was basically planned for more complex street-maps, but most of the time, it only uses the simple Manhattan structure. In contrast to its counterparts (Freeway and Manhattan), the City Section model is target-oriented with random aims. Its movements are calculated with a shortest path algorithm. If a mobility node arrives at its target, it acts like the Random WayPoint algorithm and waits for a defined time period, until it heads towards a new target. In further developed versions, the City Section can have traffic lights at the junctions, and the nodes can wait for a random - instead of a defined - time period, until they continue to move. It possesses a daytime-oriented traffic plan and has different speeds for different streets.

Graph Based Mobility Model

The Graph Based Mobility Model depends on the graph theory of computer science and uses its data structure and algorithms. The vertices have a multidimensional character ([x, y(, z)] coordinates) and the edges have defined angles. The huge number of graph based algorithms makes this mobility model very flexible. For example, many shortest path algorithms, automatic subdivision, different levels of details, mesh decimation, progressive meshes and the use of geometrical figures are integrated. But the advantage of its richness of algorithms makes it hard to control, because the use of specific functions depends on an accurate analysis of its assignments.

Functional Mobility Motion Summary

This section provides a tabular overview of the existing functions of all above presented mobility models. In this table, the mobility models are differentiated by their realistic motion, meaning, if the mobile nodes of the single mobility models follow restrictions, which create a natural course of motion, for instance, such as restricted motion in dependence on their former direction (Gauss Markov or Boundless Simulation Area) or motions depending on a given direction (Freeway, Manhattan, City Section mobility model). Furthermore, they are assessed by many more different criteria: if they are subject to geographical restrictions, if they act target-orientated, if they possess a restricted choice of direction, if prior directions or velocity influence their next moves, if they move within an unlimited area, if the motion is concluded by defined distances or time intervals, if they are graph-based, if they possess realistic velocities (such as slow accelerations when starting, or slowing down in curves) and if the nodes are allowed to overlap or not.

	Realistic Motion	Geographical Restrictions	Target Oriented	Uniform distributed Directions	distance and speed memory	Boundless	Distance and Time Break	Graph Based	Realistic Acceleration	Node Collision
Random Walk				x			x			x
Random WayPoint			x	x						x
Random Direction				x						x
Boundless Simulation Area	x				x	x	x			x
Gauss Markov	x				x		x			x
Probabilistic Random Walk	x						x			x
Freeway	x	x						x	x	
Manhatten	x	x						x	x	
City Section	x	x	x					x	x	
Graph Based Mobility Model	x	x	x			x		x	x	x

Table 2.3.2: Functional Summary of the Mobility Motion

2.3.3 Wireless Communication

In this section some popular traffic topologies are presented, which often appear in the Internet. These topologies are more complex than the already implemented traffic-generator in the NS-Mapper (Rudimentary-Traffic-Topology Plugin), which makes randomised traffic for FTP, Telnet, Constant Bitrate, Exponential, Parento and Trace Traffic.
In contrast to section Mobility Models 2.3.2, which particularly dealt with the exact procedures of the generated movements, this section focuses on the scheme of the topology, without going into details of various specifications. Usually, the protocols, which generate the traffic, need specific implementations by the used simulator. If the simulator does not support an explicit implementation of a protocol, the traffic has to be approached to the given capabilities of the simulator. That means concretely, the topology and the randomised distribution are more important for the NS-Mapper than the way how the simulator uses this data. In case the simulator is not able to support a protocol, the traffic can only be converged to the topology and cannot be analysed in all matters. However, in this thesis, we want to generate communication topologies, which are independent of the capabilities of the simulators.

Game Based Traffic

The Game Based Traffic is usually exchanged between one single server and a number of clients, which can be distributed all over the scenario. The traffic depends on the game and the category of game, which the users play, because they create different kinds of traffic at different time intervals. It is optimal if the user of a Computer Based Traffic Model does not only have the possibility to change the traffic between server and clients, but can also select a popular game from a predefined set of games.
Usually the mobility nodes, which are connected to a game server, do not disconnect from the server, if the player interrupts the game for some time. That means they receive and produce traffic, although the user does not play. It is possible to make a threshold time-value, which decides if the mobility nodes disconnect from the game-server and reconnect, when the user starts to play again.

HTTP Web Traffic

Hypertext Transfer Protocol (HTTP) is a method to transmit information via Internet. Its purpose is to publish and to make text, pictures or additional contents, for example compressed files, available. HTTP is a request and response protocol between clients and servers.
In this traffic model, the clients send a request to a server and in return, receive information in form of traffic in different magnitude classes from the server. The location, from where the requests of the clients came from, can be distributed diffusely, and the intervals between these requests are unpredictable. The connections are rather short and at the end of every server transmission, the connection is detached. The user of this model can choose the magnitude classes of the content size by himself.

Bittorrent Traffic

Some assumptions consider that the filesharing traffic has more than 70% of the world wide generated traffic. The BiTtorrent protocol is one of the most popular ones of this category. North American cable industry estimates that BitTorrent possesses 55% of the upstream traffic of the cable companies access network[17]. In contrast to other file sharing protocols, BitTorrent is published under a free license and the implementation and its construction topology is well known. Therefore, the selection fell on this peer to peer file distribution protocol.
A host computer (seed or tracker) coordinates the file distribution. A client connects to the tracker and receives information about the topology of its network. As long as there is no other participant, the client only connects to the server (tracker) and begins to request pieces of the wanted file. The protocol divides the file into a number of smaller pieces, which are transmitted.
As soon as other clients enter the network, and have enough data received from the tracker, the clients begin to exchange pieces with each other, instead of downloading the file, and its pieces respectively, directly from the seed. Clients, who have finished downloading, may also choose to act as a seed. The BitTorrent topology is visualised in figure 2.9, the tracker is connected with each client, and the clients are connected with each other.

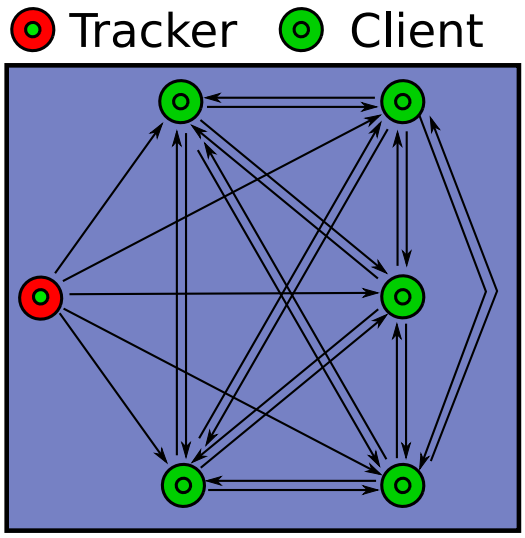

Figure 2.9: BitTorrent Topology

Video and Audio Stream Traffic

Video and Audio Stream Traffic depends on media, that is continuously received by users, while it is being delivered by a provider or another user. The name refers less to the medium than rather to the delivery method. In contrast to other classic distribution methods (radio and television), which transmit their data via multicasts, the audio or video streaming in the net mostly addresses and delivers its data to the user, who requests the media. In a simplified Media-Stream topology a client (user) requests an audio or video stream from a server by TCP and receives data from that server by UDP.

Random Traffic

Even if this category of random traffic is not used in this diploma thesis, it was presented shortly, because it is used for many other modellers.
The intention of this category is to split the traffic models in realistic pieces. In the end, random traffic is a mixture of all the above described models, and thus, there is no need for a common random traffic model, even if it is the most important one.

2.4 Chosen Algorithms

2.4.1 Node placement

In section 2.3.2 we explained random algorithms with a uniform distributed node placement, a pre-calculated motion placement and a group based placement. The uniform distributed node placement is already implemented in the NS-Mapper and will be taken up again in the comparison of the random algorithms in section 5, to validate the quality of an extended node placement. The second algorithm we do not chose is the the Gaussian distribution. This technique is too easy to implement and needs only a change of the random number part in the implementation of the uniform distributed placement algorithm. A further reason not to choose this algorithm is the missing knowledge about its realistic behaviour.
Therefore, we have to decide between two remaining strategies. As described above, a group motion strategy is a development of a single movement algorithm and extends its functionalities. But a group based node placement is only important, if we also choose a random node movement, which supports the group based mobility model. Thereby, we develop a single node motion in this diploma thesis. The choice was made for the "Mobility Motion Based Pre-Calculation" algorithm of section 2.3.1, because at first, it has the most realistic behaviour of all random start point techniques, and secondly, it can be used for every existing mobility motion in the NS-Mapper.

2.4.2 Node movement

Whereas we implemented the geometrical "Random Waypoint Mobility Model" already in a previously developed version of the NS-Mapper, this time, we decided to choose an algorithm from a different category, because the "Random WayPoint" is the most realistic mobility model of all of them in this category. That means, that we have to make a decision between a Step-By-Step or a graph based mobility model strategy.
Even if the Step-By-Step algorithms have a limited selection of directions, they can reach every position in the simulation, as long as they move within allowed areas. In Graph Based mobility models it is a little bit more complicated. The normal graph based mobility models have strictly defined edges and directions for their moves. It is surely possible to increase the level of vertices and edges in the meshes to extend the number of positions, which

can be accessed by the mobility nodes, but this enlarges the complexity of the calculations in an unpredictable way. A further problem with the graph based algorithms is, that the NS-Mapper has dynamic Movement-Fields, which enable the crossing between two different fields. That means that a graph has to recognise the other MovementField and has to merge or demerge with it. That creates two problems: A slow drive-by of two MovementFields forces a rebuilding of the graphs, as long as they move (Lookup picture 2.10 a). Without rebuilding the graph, while the Movementfields are moving, the connected edges are stretched to an unusable state. The second problem is that the MovementFields often overlap each other and so the grids of the graph overlap. The question is how to merge these two graphs. The best way is to dent the two graphs in the crossing areas to prevent them from overlaying each other (Lookup picture 2.10 b). All these calculations can be very complex and there is not enough time to implement these difficult strategies in the diploma thesis.

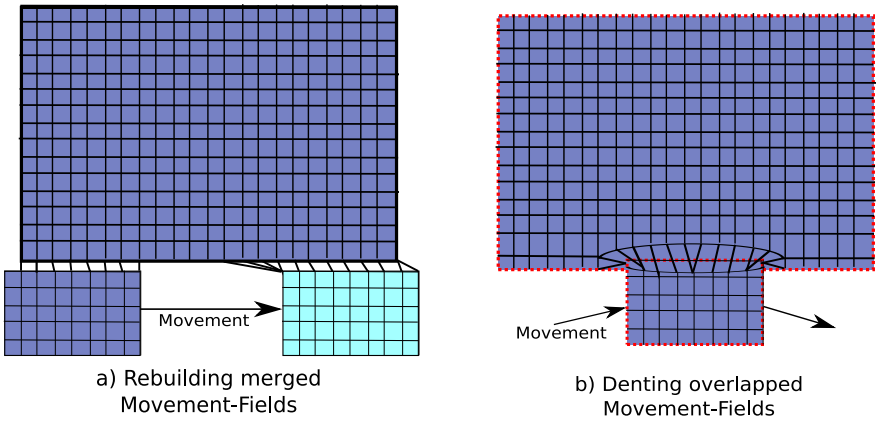

Figure 2.10: Merging of Graph Based Movement-Fields

A Step-By-Step algorithm is an adequate choice regarding the realistic behaviour of a random mobility model. The speed and the chosen direction depend on its last calculation, which guarantees a relatively realistic motion. But not every mobility strategy can be integrated into the NS-Mapper.
The limited topology of the Boundless-Simulation Area mobility model is not compatible with the construction of the NS-Mapper, because the disappearance of an unused Movement-Field out of the simulation area is a fundamental concept of the NS-Mapper, which is necessary to encapsulate different connection regions.

We decided to use the Gauss Markov Mobility Model instead of the Probabilistic-Random-Walk, because, as described in section 2.3.2, a completely natural probability is hard to find. The Gauss Markov has all capabilities to validate its behaviour in the range of the limitations, made by the NS-Mapper. Its random movement is much better than the movement of the Random WayPoint, because it possesses a rudimentary memory. But still the integration of this algorithm is not uncomplicated, because the repulsion of the nodes from the boundaries, which guarantees a soft redirection, prevents an overstepping from one MovementField to another, and this disadvantage has to be managed.

2.4.3 Wireless Communication

A decisive factor of the chosen communication topology is its easy spreading in a wireless environment. The NS-Mapper mainly generates scenarios for ad-hoc networks. These are often independent, but are able to connect to each other directly. Therefore, they do not need a base station, which connects the nodes to the internet. The support of web topologies with the HTTP or HTTPS protocol is a lot less important for ad-hoc networks, even if the NS-2 has more than one patch for this kind of protocol, which can be used within the NS-Mapper.
Other possible applications for ad-hoc topologies are game based traffic and latterly, the exchange of data, audio or video, caused by hardware, such as the Portable Play-Station, the IPod or the Zune-Player. In a previous work, the student research project, it was shown that the “Rudimentary Topology Plugin” of the NS-Mapper is able to handle the topology of game based traffic. That is why our priority is not to implement a the game based traffic plugin.
In our days, the streaming of audio and video files in ad-hoc networks is becoming more important. In the near future, it can be used to offer people information or distraction, if they are waiting in a traffic jam or for a train or a bus. But the topology is not so complex that it can not be generated with the above mentioned “Rudimentary Topology Plugin”.
The peer-to-peer applications are very popular, and this will most likely not change in the near future. There are many fields in ad-hoc networks, in which they are applicable. For example, in schools or universities when many pupils or students want to exchange data. In contrast to the data in

streams (as long as the user does not need temporary available information) peer-to-peer networks can store the data in an easy way. For example, currently the IPod and the Zune player exchange their data in a peer-to-peer way. As mentioned in the section 2.3.3, the amount of peer-to-peer data in the internet is very extensive and the mostly used one in that category is BitTorrent. BitTorrent topologies are relatively complex and not as easy to realize as the other applications.
Still, an automatic construction of this complex topology is very useful and so the choice was made for this kind of application.

2.5 Summary

The aim of this thesis is to develop better or missing modelling functions for the NS-2, that means, to improve and extend it in three categories: the node placement, the node movement and the wireless communication. In this chapter the presently available and, for the NS-Mapper compatible, applications of each category, regarding their advantages and disadvantages are discussed. It is worked out precisely, which applications are most suitable to fit and therefore, are integrated in the NS-Mapper.
For the node placement the Mobility Motion Based Pre-calculation Model was chosen, performing a pre-calculation and therefore, offering a larger variability of techniques with different probability distributions without interfering with the actual simulation or trace files.
In the process of evaluation for the node movement, we considered a large variability of mobility motion models and checked them for their capability of reproducing a natural behaviour of human beings, which in the end, most options failed to achieve. Finally, the decision was made for the Gauss Markov mobility model, which offers the advantage of having a rudimentary memory. By being capable of remembering the last direction of each node, this model is most suitable to produce a natural behaviour within the limitations given by the NS-Mapper.
For improving the wireless communication of the NS-Mapper the implication of BitTorrent fitted best, allowing the users not only to receive information from a server, but also to exchange information among themselves. Although BitTorrent topologies are very complex and not easy to realize, its advantages of being user-friendly and future-oriented predominated.

Chapter 3

Design Concepts

This chapter describes the design concepts of the plugin structure and the framework of the randomized classes, in which the NS-Mapper functionally works. The different processes of building these structures and classes are explained and their advantages and disadvantages are discussed. Additionally, this chapter addresses security aspects, which have to be considered. Furthermore, it deals with the important aspect of the reusability of the structures of this design concept. After a detailed process of evaluation, a decision is made on the basis of these information and conclusions.

3.1 Objective and Challenges

The first central aim in developing the NS-Mapper editor is to integrate the missing modelling function, finding the most suitable way to implement the chosen extensions. For a simple expansion of functionalities the scenario editor has to develop a plugin structure, which separates the developed extensions from the core framework of the program, still guaranteeing a smooth and stable working process, with an easy access to parts, which control the data structure. A plugin structure makes the development of the extended functions independent of the development of the core program, and vice versa. That is the reason why the NS-Mapper uses a plugin structure for the extended features, instead of having these extension integrated directly in the core system of the program.
Furthermore, a framework has to be developed, which supports the programmers in adjusting and translating general simulation techniques to the very

specific requirements of the NS-Mapper. This has to include, for example, the restricted movement and access to the simulation environment with its static and dynamic MovementFields. Additionally, for the developed framework and the plugin structure it is very important to design and implement a uniform integration and presentation platform for the automatic exchange of data and the interaction between the plugins themselves.
The second central aim is to establish complex scenarios for wireless ad-hoc networks in an easy and fast way, with click-and-point features and editing functions. These have to be able to cope with the sophisticated capabilities of a network simulator and shorten the time of initial training. The NS-Mapper has to give researchers the possibility to plan and organize scenarios with scalable functionalities. The scenario editor has not only to generate wireless scenarios with normal editing capabilities, it also has to support the researcher in various random strategies for node placement, -movement and -traffic. To make the NS-Mapper meet the requirements of a scalable environment, the researcher has to have the possibility to combine all different random strategies easily in one simulation without any restrictions. For this reason a strategy has to be developed, which allows the combination of different random strategies in one single scenario.
Furthermore, representative random algorithms for the three categories, node placement, node movement and traffic, have to be implemented in the NS-Mapper, which will have to demonstrate their functionality of the above described objectives. The quality of these algorithms will to be evaluated in a direct comparison with rudimentary algorithms, which were already earlier implemented in the NS-Mapper and which have to be adjusted to the new options, data and plugin structure.

3.2 Package Design Patterns

In the Java design patterns, there are two structures to access a program. The first one is well known and possesses a real directory structure and has binaries and library classes. The Java classes are not so different from the files of a normal operating system. The executable binary files are in a root directory or in a bin directory. But in contrast to Windows platforms, where the executable files possess an “EXE” extension and therefore, are easy to identify, Java does not mark its executable files. In Java an executable

file does not look differently from a library file. Its class-binaries are loaded from a virtual machine, and this special class does not differ apparently from other classes (objects or libraries). Either the user knows the class, which starts the program, or he has to search for it, by executing all classes. This problem inspired the Java developers to generate zipped program packages, called "Jar" files, which present the second way to access a program. This package hierarchy knows which class has to be started. The developers mainly provide their Java programs with these packages, because they are easy to handle for the users.

Unfortunately, the Jar packages can not be changed, if the virtual machine runs the program at the same time. This creates some difficulties for the developers if they want to load a dynamical program part in Java. A main factor of disturbance in the plugin development arises, when the program has to restart during the development process. Then, the developer has not only to recompile the modified parts, but he must regenerate the whole package again, only to reload a new version of a plugin. These operations are very time-consuming, particularly if the program is huge, like the NS-Mapper.

To bypass this problem, a program design can either absolutely abstains from Jarpackages, but on the other hand, most users like the compact files with their easy access. The second, much better way is to divide the program into a virtual core package filesystem (the Jar file) and a dynamical part, using the common filesystem. This way, the work of the developer is much easier, because he does not have to touch the Jar packages. If he wants to implement or test a dynamical program part, he only has to reload this dynamical part into the program without restarting the whole program. Thus, the developer has an easy and dynamical access to the program, and both, the user and the developer, can use the Jar packages. A further advantage of this technique is that the developer can modify his plugins, without the need of making changes in the core program. Thereby, the developers of the core program and the developers of the plugins can work independently from each other.

Regarding the design patterns, the NS-Mapper will use the method of the two filesystem parts. It assigns the core program and the core plugins to the virtual Jar filesystem but all extensions to the real filesystem. Thus, there will be two different routines for accessing these program parts, and

two different ways to access configuration files, which merge the registration of plugins.

3.3 Datastructure Requirements

There are many ways to exchange data within a program. The former NS-Mapper version 0.1.* used the easiest way. The various parts got their data by simple method calls. But as the program has developed and the interactions between the parts and especially the plugins have become more complex, the implementations are prone to error, and the changes are expensive.
Especially for plugin based programs it is more practical to reverse this process and to not give the data to the plugins, but to fetch them from a central point. This makes changes more flexible, and the plugins and data structure is safer, because the maintenance of the different program parts is much easier.
There are two possibilities to administrate the main data structure. The first way is to use a Remote Method Invocation (RMI), which makes data available from a local or remote socket. It uses an exchangeable object within the program. The second possibility is to use a static data structure, which makes sure that there is only one reference to a specific data block. Thereby, a static path is available, from which data can be fetched and stored.
The main problem with the RMI is that the exchanged objects only look like local objects; but if the data is transmitted by the socket, all references to other objects are destroyed. For example, if you get the same data part twice with the RMI, they are not the same objects anymore. Now they are two different objects with the same data. If this object is send back to the central data structure, it is very difficult to rearrange the data to the central structure again. In case the transmitted object is a GUI element, the rearrangement is actually impossible. But the NS-Mapper uses some dynamically generated GUI objects. This is the reason why the static reference should to be used.
But there are also problems with static references, which RMIs do not possess: and that is the old problem with the security. Using a static reference, every plugin can access the data structure without any obstacles or problems. Although there are data structure administration interfaces, which

test the regularity of values and objects, the program is not save anymore if the developer chooses to ignore it. For the NS-Mapper we choose the minor problem and use a static reference in the main data structure. The implementation of an RMI to cover all possible errors is very complicated and too time-consuming for the limited time frame of a diploma thesis.

3.4 Plugin Design

There are many ways to load a plugin dynamicly, but only a few of them are practical and reasonable. It has to be decided from case to case what the best way for a program is. Here is a short enumeration of practicable ways:

Unknown Reflections An arbitrary class is loaded without knowing its structure. The class is examined with the reflections API, which gives the developers information about it, such as its method names, their parameters and return values. If the developer wants to invoke a method of this class, he has to analyse this information and build a dynamical method call.

Interface Reflections The dynamical loaded class has to implement a specific interface. With this interface, the developer knows the structure of the class, he wants to load. After loading it, he can downcast the class to the interface structure and use it like every other object. The interface defines only the structure of a class, but does not have any own executable code.

Abstract Reflections The dynamical loaded class has to extend a specific abstract class. With this class, the developer knows the structure of a class, he wants to load. After loading it, he can downcast the class to the abstract class structure and use it like every other object. The abstract class can have executable codes in contrast to the interface. This means the abstract class can have fully implemented methods. If an abstract class wants to define only the method name, the method must be declared as abstract.

As mentioned above, in Java it is possible to detect and load program parts automatically, even without knowing their object structure. But to

identify and use an arbitrary class as a regular program part, all plugins must be loaded into the memory first. They have to be loaded, even if they are not used, because the program is not able to detect if a class in a directory has a specified plugin structure or if it is of a different class, unless its structure has been analysed. This preload of program parts can require decisions of how to use the memory. It is sometimes difficult for the common user to determine if a program has low or high memory costs and therefore, what the proper load for this program is. Additionally, in Java it is more complicated to determine the total free memory.
Java runs its programs in a virtual machine and the programs demand their memory from the operating system at their start. Thus, two memory problems exist: The first is that the program is not able to extend its memory in a running status. If the user makes some memory-consuming tasks, the program crashes and all data is lost. The second problem is that Java reserves memory from the system and references it to its working stack in the range of this memory. The free memory is arranged between the instantiated memory of the stack and the whole reserved memory. Thus, the program can not exactly determine whether there is enough space for calculation, because the reserved memory of the operating system is occupied, but the instantiated stack is empty. A warning from the program is not really reliable. The best way is always not to waste any memory.
Alternatively, one can load the plugins and, after inspection, delete them again. But apart from the view that is not very elegant, this procedure is time-consuming. Additionally it not sure if the garbage collector deletes the plugins at the right time. A Java developer informed in the “devradio” show[6] that a garbage collector can clean its data immediately, but also never. The above explained sequences are relatively easy to handle, but they are not the proper way for the NS-Mapper.
In the former NS-Mapper versions the plugins were automatically recognized, loaded and reloaded. To make the plugin work safer, we made a step back in that structure. The developer has now to pay more attention to the plugin certification. We decided for using a procedure the way the Eclipse-IDE registers its plugins. All plugins are certified in config files and the NS-Mapper loads them only if they are needed during the working process. The config files are analysed at the startup time and only the names of the plugins are included in the menu structure and the structure of the random

process.
The NS-Mapper never controls the status of the plugins, that means that it does not control if a plugin is recompiled. The developer has to reload a new plugin himself if the version changes, but he does not need to reload the entire program.

3.4.1 Plugin Hierarchy

As described in section 3.4, Java provides the possibility to detect and load dynamical objects from classes, which can be executed. To achieve that, the Reflection API[20] offers some procedures, which investigate what kind of data calls a class needs (namely the constructor and its methods) and it can choose their data structures from a known pool to invoke them. These are features, which could be realized for designing a plugin structure. On the one hand, it would make the concept of the plugin invocations very customizable. But on the other hand, this independence does not meet our high expectations of support for the NS-Mapper, because the developer could generate every guideline in his plugin, which he intends. That is the reason why we rather want to integrate the programmer in our concept of the NS-Mapper, as already described in the previous section.
We want to realize an interface, which guarantees the compliance with our regulations. The Java API offers two solutions how a class for the plugin structure can be implemented. The first possibility are the “Java Interfaces”, which create an interface frame of a class, lookup section 3.4. All methods, defined by a Java Interface, are abstract and have to be overwritten, because this frame of a class does not allow any method bodies. One single class can implement more than one Java Interface. The second possibility is to define an abstract class. This abstract class is able to mix real implemented methods and abstract methods without a body. Here, a class can only be extended by one abstract class, in contrast to implementing an interface.

As shown in the class diagram 3.1, we combine the interface and abstract classes in our plugin concept. We want to make sure that the reusability of this structure becomes an important feature, because we want to integrate this class hierarchy for every plugin used in the NS-Mapper. For that, we separate the codeless interface structure (PluginInterface) from the abstract

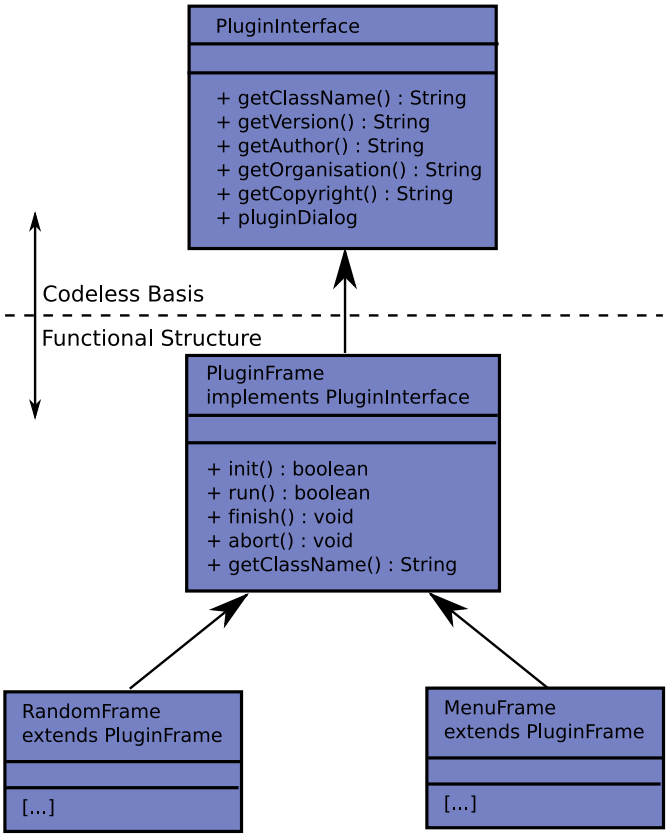

Figure 3.1: Plugin Structure

class (PluginFrame) with helpful method implementations. The reusability guarantees the developer, that he can integrate his own implementation in every program part of the NS-Mapper and that he can use processes of other plugins for his own extension. We will make use of the abstract construct as often as possible, because this concept gives us the possibility to support the plugin developer in recurrent tasks. Additionally, we can enforce him on our supporting concept and guidelines of how a plugin has to be used.
The basis of the class diagram 3.1 is the PluginInterface. This Interface determines that the PluginKeeper, which stores all loaded plugins, can find them again. It also offers some information about the developer. The PluginFrame, which implements the PluginInterface, determines mainly the sequence of the init, run, finish and abort methods, described in the next subsection. The PluginFrame is used by every implemented plugin, which are all plugins, that are integrated in the menu or in the random plugin

structure of the NS-Mapper. The improvement of the menu structure is adjusted to the MenuFrame and the adjustment of the random plugins is done in the RandomFrame. In the next subsections the different generics of the classes for the plugin structure, their tasks and their execution are described in detail. The RandomFrame is part of the random algorithm concept and will be described in the section 3.5.

Plugin Interface

The object oriented programming guidelines recommend to group functionalities as often as possible. The PluginInterface sequesters all those methods, which must be overwritten by the plugin developer. That is why we use an interface and not an abstract class. In this case, the base functionalities describe organizational parts of the plugin.
The base interface of the plugin structure is the "PluginInterface". With the abstract class "PluginFrame" the interface is used by menu and random plugins in a similar way. It contains some organizational parts for collecting information about the developer, such as author, organization and e-mail address for bug tracking contact, as well as the license and the version of the Plugin. The emitted information gets back in a String and can be used in a plugin, which collects and displays this information. With the pluginDialog method the developer can generate additional information about his program, for example, providing a small help, etc. Thus, he can define his own java dialogs or frames by this method. The NS-Mapper does not interfere with the representation of the author´s pluginDialog, therefore it is void.
The method getClass() is used for the identification of the PluginKeeper class (lookup 3.4.2), which stores all temporary plugins and makes sure that a plugin is only loaded once. The developer of a plugin does not have to pay attention to the method because it gets its default implementation in the abstract class: PluginFrame. It is first declared in this interface because the method is often used for the uniform raw downcast by the PluginLoader and some menu plugins.

PluginFrame

The abstract PluginFrame class implements the init, run, finish and abort methods and forms the basic construction for the functional part of the plu-

gin. As detailed described in the next subsection 3.4.2 and figure 3.2, the init method prepares the plugin execution and detects, if the plugin can work without problems. The run method implements the functionalities and perhaps the manipulations processes to the data structure. And the finish and abort method completes the plugin execution. It implements a default method body in the init and run method, which forces the user to overwrite these methods. The default return value of the init method is the boolean value: `false`. The user has to implement the body of init to allow access to the run method, which also returns `false`, to guarantee that the PluginStarter, which starts all plugin, invokes the abort method by default. Another default implementation is declared for the pluginDialog method, which generates a raw blocking java dialog with all above described information (author, organization and e-mail address). And finally, the getClass method declares the simple class name, which distinguishes a single plugin class from other plugins in the PluginKeeper, which stores them. The class name is defined by the Java-Reflections class, which is implemented in the Java API.

3.4.2 Common Execution of a PluginFrame

There are two main ways how to design a plugin execution. At first, a single entry method can be defined. By means of this entry point, the execution of a plugin is not specified at all. That means, its execution is not controlled by any instance, but runs automatically. The second way is to define an execution hierarchy, in which a program and the called plugin pass more than one phase or sequence. In every phase the plugin has to carry out specific tasks to support a smooth execution. The problem with the second way is that this hierarchical concept provides only a guideline. However, we chose to design the NS-Mapper plugin structure by using the second way. Therefore, we defined processing guidelines for the plugins, expecting the developer to interfere in these processes.
The plugin concept of the NS-Mapper is modelled after some parts of the Eclipse-IDE plugin structure[25], as for example, used in the API org.eclipse.ltk (Refactoring-API)[10]. The sequence diagram in figure 3.2 shows the order of events, which are necessary to support a relatively safe procedure, which loads, stores and executes a plugin.

When a program part of the NS-Mapper wants to start a plugin, it in-

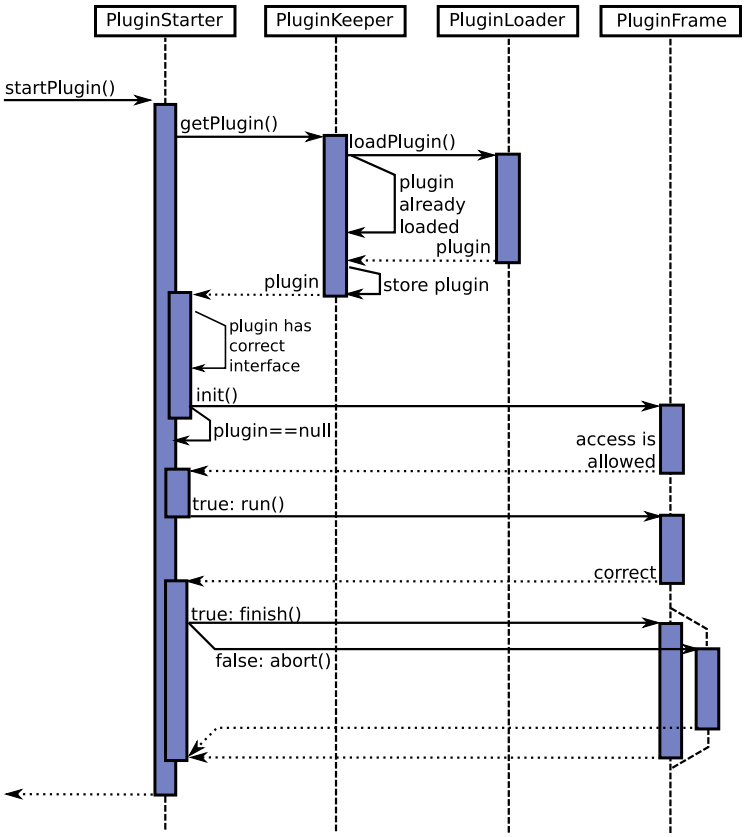

Figure 3.2: Sequence Diagram: Start of a Plugin

vokes a PluginStarter class by its name, refer to sequence diagram 3.2. The PluginStarter coordinates the correct order of how a plugin is processed. At first, it calls the PluginKeeper class by the name of the plugin object. The PluginLoader checks its database for the name of the plugin and if the plugin was already accessed by a previous invocation. If not, it starts the PluginLoader, which loads the plugin into the memory. The PluginLoader returns the object and the PluginKeeper stores it in its database. Till that time point, the reached object was not controlled if it is a real plugin, because the PluginKeeper does not know how plugins looks like. It is therefore possible that it has to access the plugin again, especially if it does not have a correct interface frame. Then, the PluginKeeper returns a reference of the object to the PluginStarter. The PluginStarter tries to cast the object into its used plugin interface. If the object does not match the interface, the PluginStarter overwrites the object with null, indicating that it does not have a

content anymore. If the cast was successful, the PluginStarter passes three sequences, which represent our guideline for the developer. These sequences are pre-determined and have to be implemented by the plugin developer to execute the plugin. The sequences consist of: the initial condition, the main manipulation and the exit section. This three sequences represents the functionality of a plugin, in which for example the "Gauss Markov" algorithm can be executed:

1. When the PluginStarter invokes a plugin, it checks the initial conditions and collects information about the calculation or the manipulation process. For example, if a plugin wants to calculate random start point positions for mobile nodes, an initial condition is: Are there any nodes to calculate? If yes, the information is: how many nodes. This information collection is performed by the init() method. So, it is a fast inspection of rules to make sure that the plugin can work correctly and smoothly in the temporary context. If there are any conditions which thwart, it will leave the whole course of events. Optionally, the initial process might also try to call the attention of the user. To state an example: in case a random plugin wants to generate the start point of the nodes, but the user has not defined any nodes yet, the plugin process will stop immediately.
   Only if the workout is correct, the main program will start the manipulation sequence. The return value of the init-method is a boolean, when all initial conditions are defined and the method comes to the decision, that the run method can do its calculations without any errors, it returns `true` for a right decision, otherwise `false`.

2. The entrypoint of the algorithm section is the run() method. All calculations and manipulations, depending on the data structure, will be done on the basis of the conditions and information given by the init() method. The run method calculates all necessary processes and manipulates, if needed, the ValueTable class, which keeps the main data structure, and especially the MovementField data structure and RandomNodeOptions (default and extended options of the random-algorithm-part) and optionally, interactions with the user.

3. The exit sequence is divided into two parts: The finish and the abort

method. If the run method leaves in a clean status, the plugin decides to finish the manipulations. If something goes wrong in the workout, the plugin decides to call the abort method, which cleans or undoes the calculations within the temporary data structure, thereby, the plugin can leave in a regularly status. When the run method leaves, it returns a `true` value to the main program for the finish() method, otherwise `false` for the abort() method.

In contrast to Eclipse, the NS-Mapper does not allow the plugins to pass these three sequences on their own. It determines the programmer to use the PluginStarter class, which controls the plugin sequences. Eclipse only suggests this to the developer, to give the various operation areas more independence, whereas the NS-Mapper wants to guarantee more security. The developer is requested to think about the initial conditions and the finish and abort processes, because he must overwrite these methods, when he wants to integrate a plugin into the NS-Mapper.

Usage of the PluginFrame

As pictured in image 3.1, the PluginFrame is divided in two categories, which can be extended from this PluginFrame. The first is the "RandomFrame", which provides the basic structure of a Random Plugin, such as of a "Start Point" or "Mobility Model" Plugin. This will be described subsequently. The second is the "MenuFrame", with which a plugin can be designed, which is integrated in the menu structure of the NS-Mapper.
Both categories, RandomFrame and MenuFrame, demand different requirements from their surrounding. That is why both types of plugins had to be separated from each other. However, we do not want to refer to the MenuFrame, because this is not part of the diploma task.

3.4.3 Reusability of the Implementation

The basic declaration of the plugin structure is kept very simple and should cover all primary requirements of the reusability. It is the lowest common denominator, which can be used from the random and the menu plugins together. This plugin structure secures that the different plugins can call each other, if needed, because they can use the same PluginStarter to access

their init, run, finish and abort methods.

3.4.4 Enhancement

The use of plugins in the menu and random sections does not contain any restrictions regarding the extensibility of functions and random based algorithms in the NS-Mapper. All ideas are in the hand of the developer. But there are some parts in the course of events, which still could be improved. As discussed in section 3.3, there are some considerations about the use of RMI and static references of the data structure, regarding the resulting security. In some cases it is better to use RMI, although the effort of reintegrating data in the data structure is much higher. But in the end, it is absolutely practical and much saver for the data structure to use it for the core data in the ValueTable (main data structure) object, which contains the ValueNew, Node and MovementField objects.
From a general point of view, it would be desirable if the whole NS-Mapper was a plugin itself. The core program would only start other functions and coordinate program parts, like in the Eclipse-IDE. The upgrade of functions, for example in the PaintingArea or the Choicetree would be more flexible and autarkic. But at the current level of this software, this is not obligate. It would only add some kind of elegance. However, also in the described form the plugin structure is realized and already clearly improved.

3.5 Randomized Algorithms

In the NS-Mapper the frame layer separates the definition of the random plugins (RandomFrame) from the definition of the menu plugins (MenuFrame) lookup section 3.4.2. In contrast to the menu plugins, the random plugin structures perform the declaration of the nodes, improving them regarding their random start point, random motion and random traffic. The random plugins are the main plugins, which have to be extended in this diploma thesis and are described in this section, whereas the menu plugins will not be described further.

3.5.1 Random Plugin Structure

The class diagramme in figure 3.3 represents the declaration of the random plugin structure. It shows the RandomFrame, which extends the PluginFrame (lookup figure 3.1 and section 3.4.1). This increases the functionality of this plugin category in regard to the special needs of the random workflow. These are mainly the implementation of threads and an automatic framework to declare default values and to process them. Additionally the diagram shows the RandomMovementFrame, which extends the RandomFrame. RandomMovementFrame is an extension only for the random movement plugin to access the processing of a single wireless node.

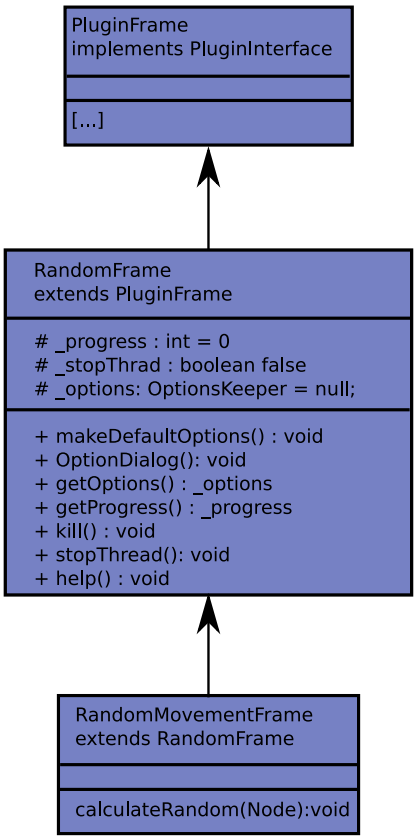

Figure 3.3: Random Frame Plugin Structure

The declarations and functionalities are described in the following subsections.

RandomFrame

As described in the previous section, the RandomFrame is a layer, which separates the menu plugins from the random plugins. The random plugins represent the structures of the random starting point and the random traffic plugins. In contrast to menu plugins, they work in their own threads, because runtime and complexity of this plugin part can be very high. In that way, they can signal their temporary working status to the user or inform him that the program is already in a regular status. For this, the RandomFrame has the method getProgress, which offers the developer the declaration of an integer value for the definition of the present calculation status. This value can be given to a Java ProgressMonitor, a Java ProgressBar or an AttentionDialog. Allowed values are from zero to 100 per cent.
In some cases the user wants to abort the calculations of the RandomFrame, because the algorithm of the plugin is in an endless loop or the user is not interested in the results anymore. Therefore, he declares the kill and stopThread methods. Of course, the NS-Mapper can not always guarantee a short course of working time or the absolutely error-free sequence of an extension plugin. That is why the user must have the possibility to interrupt calculations of a plugin. For a clean ending the progress monitor can call the kill method of the RandomFrame, which sets the _stopThread boolean value to `true`. When the developer has a good position for the _stopThread boolean request, the RandomFrame can shut down in a clean way by the stopThread method.
The option methods are the most important functions in the RandomFrame class. When a random plugin is loaded for the first time, it starts the makeDefaultOptions method, which has to be overwritten by the user. This method defines all used data, which is necessary for the calculations of the defined algorithm, and declares default values for the OptionsKeeper. The OptionKeeper is described in detail in section 3.5.2. This class administrates the data and prepares automatically a user-friendly dialog, based on the data structures to manipulate them directly.
The getOptions method gives the validated values for the Node data structure in an OptionsKeeper object back or, if needed, the default values of the random plugin. Every node in the ValueTable data structure receives its own option version or a reference if the nodes are grouped.
The getHelp method is defined for offering a little help dialog to the selected

random algorithm, for example in a Java dialog or frame. It can explain the permitted and used values, which the plugin needs for its calculations.

RandomMovementFrame

The random plugins for the mobility motions need an additional method for their sequences. For that, the RandomMovementFrame class is defined. The regular run method, which calculates the motion of all nodes in a plugin, is not sufficient. The chosen concept for the node placement, the "Motion Based Pre Calculation", which is described in section 2.3.1 and 2.4.1, needs more flexibility instead of just processing the wireless nodes globally. The random node placement concept begins with the calculation of single nodes, defining them to use a special start point. But in the end, the node definition of the mobility model can differ from the start point, calculated by the pre-calculated motion mobility model. To give an example: The random starting point pre-calculation can be processes by the Gauss Markov algorithm, but its real mobility model must be processed by the Random WayPoint. In other words, the plugin has the ability to change the mobility model of a node temporarily to another mobility model.

3.5.2 Random Options

For a uniform declaration of default plugin options (lookup section 3.5.1) and their processing, the NS-Mapper needs a framework, which guarantees a standardized access to the defined values, which are valid for more than only a single plugin. For this requirement we have to define an OptionKeeper class, which contains a whole bunch of different data structures and which return their different values in a uniform way. Furthermore, the manipulation of these values should be prepared as an automatic generated GUI, which can be accessed by all plugins, that are necessary for a manipulation process.
In the class diagram of figure 3.4 we have defined how a structure of an OptionKeeper class can be realized.

The OptionsKeeper stores the operation data in an ArrayList with a type safety object, called the RPluginOptions, which defines a uniform access to the following data structures: byte, short, int, long, float, double, char, String, boolean, byte[], short[], int[], long[], float[], double[], char[],

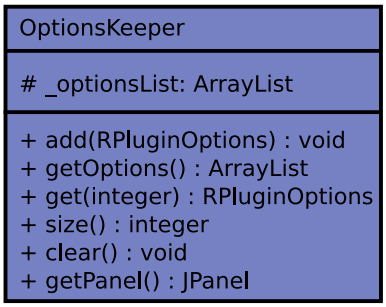

Figure 3.4: OptionsKeeper Class Diagramm

String[] and Object. These data structures are the most important ones. If the option values need a more complex data structure, it can be downcasted to an object or an object array.
In addition, the OptionsKeeper declares an add method, which adds a RPluginOptions object with the specific data to the ArrayList. The getOptions method gets the ArrayList with all RPluginOptions back. And the get method, which is called with an integer value, gets a PluginsOptions object to the indexed position back. The size method declares the number of RpluginOptions, which are stored in the OptionsKeeper. The clear method deletes all values in the OptionsKeeper. Finally we defined a method, called getPanel, which returns a JPanel, where the user can manipulate all stored values in the OptionsKeeper class. How the GUI has to look like is shown in figure 4.3.

3.5.3 Execution of a RandomFrame based PluginindexRandomFrame

To illuminate the course of events, as pictured in the sequence diagram 3.2, we want to specify the process for a RandomFrame, as represented in figure 3.5. To simplify the image, the getting of the plugin from the PluginKeeper is only indicated. The PluginStarter checks if the random plugin, which is called for execution, was already used before. If not, it starts the makeDefaultOptions method. This method initializes all values in the plugin. After that, the PluginStarter starts the init method of the random plugin. The init method collects all information about the course of events, if it comes to the conclusion that the plugin can be executed correctly. In case the process takes long, the init method can additionally start a ProgressMonitor, which

provides the user with feedback information, regarding the current state of the plugin execution. Afterwards, the init methods stops. If the init method was completed successfully, the PluginStarter calls the run method. This method performs calculations of the plugin, such as for example the Gauss Markov algorithm. During this procedure, the run method can repeatedly deliver a status report of the events to the ProgressMonitor. When the run method concludes, it reports if the course of events was successful, which causes the PluginStarter to start the finish method. If it was unsuccessful, it starts the abort method. These methods finally complete the execution of the random plugin, including a completion of the ProgressMonitor and for example the deletion of temporary variables. The case, that the user aborts

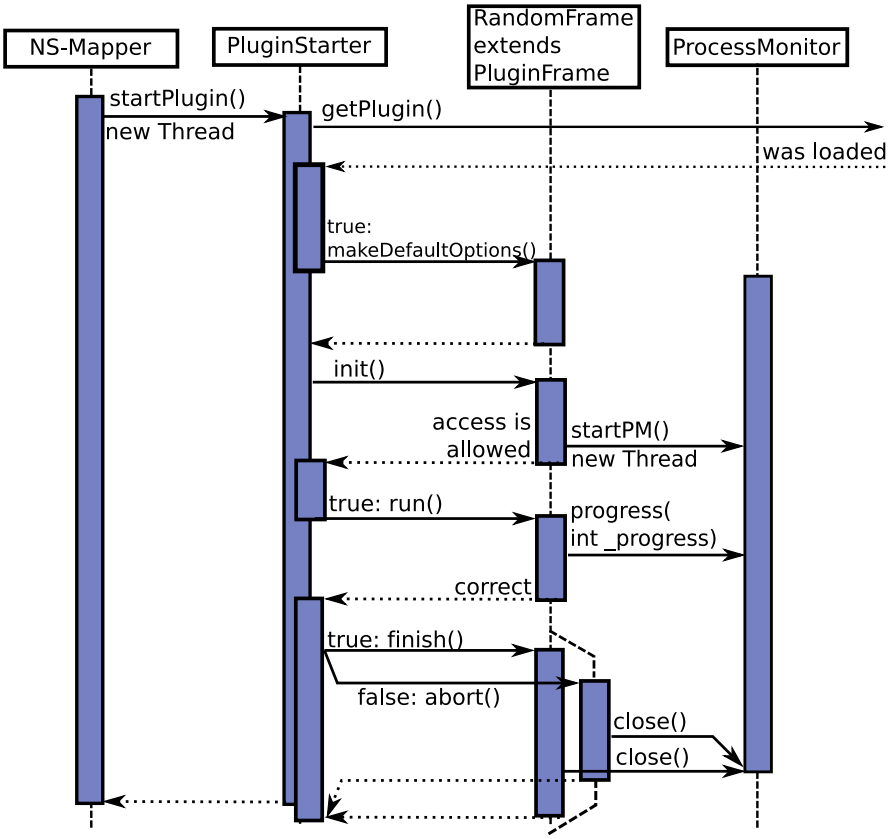

Figure 3.5: Sequence Diagram: Start of a RandomFramePlugin

the execution of the plugin and thus, the NS-Mapper calls the PluginStarter to kill the presently executed plugin is not illustrated in the sequence diagramme.

3.5.4 NS-Mapper relations

The NS-Mapper has a different environmental setting than most algorithms for mobility models. In contrast to usual simulation areas, which have an unlimited motion in their simulation environment, the NS-Mapper restricts this motion to its MovementFields, which are rectangles or polygons. The wireless nodes can not leave the boundaries of a MovementField. But the MovementFields can move on their own. That means we need a procedure, which translates the technical features of a mobility model to the technical features of the NS-Mapper.

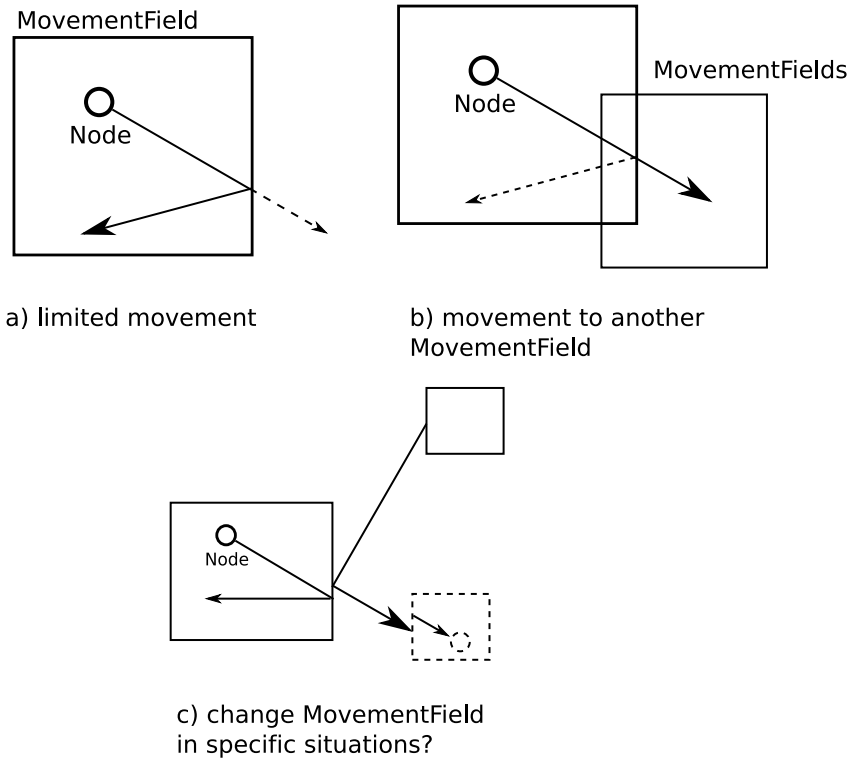

Figure 3.6: MovementField limitations

We have to define some situations, which can appear, when a wireless node moves on a MovementField. Some cases are presented graphically in figure 3.6.

Situation a) The node has to turn at a border of a MovementField and subsequently, continues moving in a different direction.

Situation b) If a node reaches the border of its MovementField, it has to pass the border, in case another MovementField touches. Thereby, it changes its MovementField, not changing its initial direction.

Situation c) The node reaches the border of its MovementField and a different MovementField is at the same position but only for a limited time. The node has to check if it reaches the border at the same time, the other MovementField touches. If the check is positive, the node has to pass the border and to go on with the same direction. If the check is negative, it has to turn around within its initial MovementField.

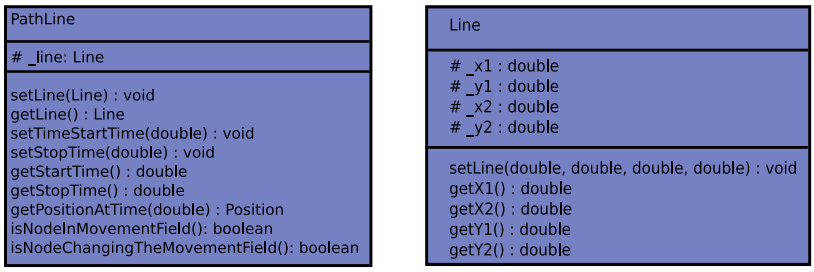

Figure 3.7: PathLine Class Diagramm

We have defined a class PathLine, which gives us the capability to calculate these problems. The class diagram 3.7 depicts its structure. We have an object Line, which represents a line with two positions ((x1, y1) and (x2, y2)). PathLine includes the setLine method, which stores the line object. Then we have the setStartTime and setStopTime method, which represent the start and stop time. With the start time and the start position of the line, as well as the stop time and stop position, we can calculate every time and every position from the beginning till the end of the line. The algorithm in PathLine proceeds the following steps:

Step 1 The PathLine has a start and stop position and a start and stop time.

Step 2 It collects all intersections with MovementField borders between the start and stop position.

Step 3 It calculates the time when the line intersects the borders.

Step 4 It starts a loop, which passes through all intersections.

Step 5 It is checked, if the line reaches a border at a specific calculated time point. If the node cannot pass the border, it sets the stop time and position to the time and position of the intersection and stops. If the node can pass this border, it starts again with step 5.

The line and time, which are returned by the getLine and the getStopTime method, have the corrected stop position and stop time, whereby, the node has its first motion segment.

3.5.5 Reusability

As described in section 3.4.3, we remarked, that the base structure of the PluginInterface and the PluginFrame has an optimal reusability. But the special requirements of the extended structures are a little more complicated. The specialization of the RandomMovementFrame shows (lookup 3.5.1) that the whole calculations and their results do not only depend on the requirement of their own plugin category. They also depend on the conditions of other plugin categories, because they have to interact with each other. It is hard to figure out, if other plugins need more access points to a specific category. In the end, it will be only possible to answer this question, when we have evaluated all algorithms in all categories for verification. But at this stage, we only have time for two per category.

3.5.6 Enhancement

Many abstract program structures have to prove their usability for the first time, when they are implemented, even if everything is planed precisely. Therefore, they should be prepared for all known cases. As described above, the random movement interface (RandomMovementFrame) structure needs an extended frame with an extended functionality for the “Motion based pre-calculation”. It is possible that there are situations, which need even more capabilities than the RandomFrame offers now. If the NS-Mapper has more than one random plugin interface structure, it changes the PluginStarter for every specialization of the interface, because it has to recognize

all changes. But it is better, when the PluginStarter can handle all possible plugin structures dynamically. This capability can be achieved with the Java Reflections API. The PluginStarter can be enhanced in the dynamical recognition of RandomFrame based interfaces, which have advanced functionalities.
In section 3.4, we recommended avoiding wasting memory, because Java can only define its free space by calculating the non-instantiated stack length. It is possible to extend the PluginKeeper (lookup section 3.4.2) and to send this class a warning, if Java instantiates its last stack space. Then the PluginKeeper can delete all those plugins, which are not used for a particular time period. This could protect the user from program crashes.
The OptionKeeper class can also be extended by more specific values. At this planning stage the OptionKeeper is only able to store single values or an array. But often the values are interacting with each other. For example, if we have a minimal or maximal value, we have to detect if the minimal value is really smaller than the maximal value. The OptionKeeper does not pay attention to its values. But this can cause errors in the calculation, if the user does not avoid them in his own program part. At first glance, this does not seem problematic. But we want the OptionsKeeper to generate an automatic GUI to manipulate these values. If the OptionKeeper regulates the manipulations of the values, the verification of the values can only be done after the manipulation process. Unfortunately, in this diploma thesis we do not have the time to validate those problems for the OptionsKeeper. Nevertheless, this has to be done in future, because there can be a huge pool of those complex problems.

3.6 Summary

In this chapter we described various design concepts of the plugin structure, regarding their usability. We mentioned the waste of memory in the plugin design, which caused us not to preload all plugins at the starting process. We discussed the access of a plugin to the data structure, and decided for a static central management, instead of a RMI based structure. We described the possibilities how to generate a flexible plugin structure, and decided for a combination of Java Interfaces and abstract classes.

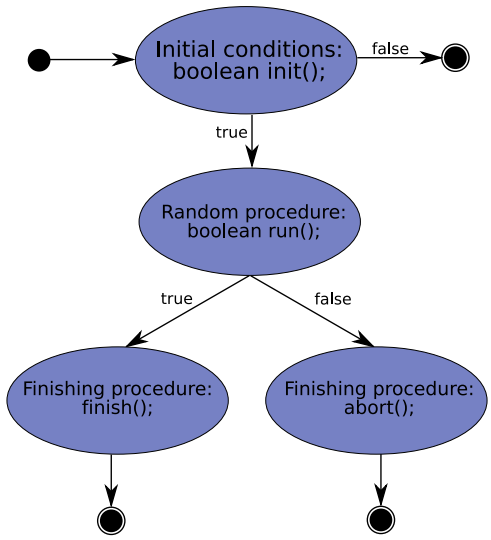

Figure 3.8: Basic plugin-structure (PluginFrame.java)

Additionally, we stated that it would be good for the programmer, if we run the execution of plugins through three states, shown in figure 3.8. The initial condition, which allows the access to the run method, the random procedure, which decides if the calculations is valid or not and finally invokes the finish or the abort state.
Furthermore, we described that the different plugins can all use the same plugin structure. It is of no importance if it is a random start point plugin, a random movement or a random traffic plugin; it might even be a menu plugin, because they all depend on the same plugin base, called the PluginFrame.
Finally, we defined an OptionsKeeper, which generates a uniform value storage for the interaction between different plugins, providing the ability to manipulate its data in an GUI based dialog.

Chapter 4

Implementation

In this chapter, we will first present an overview of the newly implemented parts of the NS-Mapper, which were necessary for the adjustment of the new functions and plugins. Afterwards, we will describe how the data structure is constructed now, how to exchange data with it, what functions it possesses and how to use them in the plugins. In detail, we will explain the concept of how plugins declare their own default data by using the OptionsKeeper, how they are now administrated in a standardized way and how they are prepared. Additionally, we will explain, how the OptionsKeeper simplifies the interactions and the exchanges between the plugins themselves and the core program. We will explain the Pathline class, which now presents a fundamental element of each random based "Mobility Motion" plugin. It provides the programmer with an easily usable way to make single mobility nodes cross from one MovementField to another, respecting the time component of dynamic MovementFields, which themselves are able to move within the scenarios of the NS-Mapper, refer to 2.2. Finally, we will deal with the implementation of the random plugins. It will be described, how they were basically integrated in the setting of the NS-Mapper. After a general presentation of this integration, we will explain its realization for every random plugin (random start point, motion and traffic) separately. It will be specified which options the plugins use, how they are constructed, and which adjustments were necessary to make them work properly in the NS-Mapper.

4.1 Implementational Overview

In this section, we can only provide a rough overview of the implementations, which had to be performed during this diploma thesis. A complete insight of all performed changes is unfortunately not possible, because the NS-Mapper had to be completely newly constructed and programmed during this time, to realize the integration of our plugin concept in the data structure of the NS-Mapper. Therefore, for the explanations we will have to concentrate on the parts, which are most relevant for this thesis, because now the program consists of more than 20.000 lines of code.

4.1.1 Introduction

The NS-Mapper consists of two parts. On the one hand, a console based program part exists, without a graphical user interface and the GUI based part. The console based part processes already constructed and saved scenarios to change and duplicate them and to export them to the NS-2. It is able to recalculate single aspects of the scenario, as for instance the random based parts, such as start point, movement and traffic. Additionally, it can adjust single code parts of an automatically generated tracefile, such as the routing protocols or the output into a nam file. In the GUI based part of the NS-Mapper, the scenarios are modelled, processed and saved.
The following description of the implementation of the NS-Mapper will only deal with the GUI based part, because this part is relevant for this thesis. To avoid confusion, we want to note that the GUI based part is called "GUIMain". From here, for example, the data structure can be invoked. However, this name does not indicate that it contains the GUI elements of the NS-Mapper, it is only the root of this program part.

4.1.2 Implemented Parts

Here we want to provide an overview of the program parts, which primary had to be programmed for this diploma thesis and which will be subsequently illuminated more precisely. These include:

DataStructure It was implemented to provide all components of the NS-Mapper with a central access point to all data and functions, independent of the place, from where they are accessed. It administrates

basic information of the scenarios, the access to mobile nodes and MovementFields, GUI components, mouse and keyboard events, etc.

OptionsKeeper The OptionKeeper was developed, because the plugins exchange data between each other, prepare them graphically and process them. It provides standardized declarations and access to data and can categorize this data and prepare it in a GUI, without requesting the intervention of a plugin developer. Therefore, plugins can coordinate other plugins, can portray and change their data, as for example necessary when using the "Motion Based Pre-Calculation" start point plugin.

PathLine This class regulates the movement of the mobile nodes and the dynamic MovementFields. It detects when and where an element is located and, depending on these calculations, examines and corrects the movement paths in the time context of the scenarios. For example, it recognises if a mobile node can pass from one MovementField to another, and if necessary, devides them in single section.

Random Plugins This implementation part concerns the single random plugins, in detail, the "Motion Based Pre-Calculation" start point plugin, the "Gauss Markov" mobility motion plugin and the "BitTorrent Topology" traffic plugin. Its subsequently provided description will contain their construction and what changes were necessary to adjust them to the capabilities of the NS-Mapper.

4.2 Data Structure

In section "Objects and Challenges" 3.1 and "Datastructure Requirements" 3.3, we have already described briefly the structures and conditions, with which data of the NS-Mapper can be accessed. Now we want to list the most important parts of this structure and explain their tasks.
As mentioned previously, the main data structure of the scenarios possesses a static reference in the main part of the program. That means that there is only one location of data in the whole java virtual machine, which can be accessed from every class, that acts within the same machine. The various parts can be called from the mapper.guiComponents.GUIMain class. It contains some general methods, which are able to reach the different objects.

Table 4.2 presents a short summary of these objects and illuminates how the single parts can be called from the GUIMain class. Section 4.2.1 and section Framework 4.3 will provide a detailed description of this list. Additional information about special method calls, which are not explained here, can be found in the JavaDocs of the NS-Mapper.

Method	Return Value	Description
getDataStructure	ValueTable	The ValueTable administrates the list of Nodes and MovementFields with all information about them, such as the positions and the paths, as well as the information about the scenario itself, such as simulation time and length.
getGUIComps	GUIComponents	The GUIComponents store all graphical user interfaces, such as the PaintingArea, the Choice-Tree or the TimeSlider.
getOptions	OptionsKeeper	The OptionsKeeper stores additional information about the NS-Mapper, such as the path to the icon and configuration directories, and temporary values, such as the name of the opened file and if it is stored.
getPluginStarter	PluginStarter	The PluginStarter coordinates the loading, the storing and the execution of all Plugins.

Table 4.2: Accessing the Datastructure from the GUIMain class

If the developer wants to work with the data structure (lookup Listing 4.1), he only has to import the GUIMain class into his own plugin, which he wants to implement, see line 1. In Listing line 2, a ValueTable is stored in the object **vt**, which is returned by the GUIMain method getDataStructure. To receive all needed data, the developer only has to call a method of listing

4.2 by prefixing its name.

Listing 4.1: Call GUIMain methods

```
1 import mapper.guiComponents.GUIMain;
2 public class PluginExample {
3         public boolean run() {
4                 ValueTable vt = GUIMain.getDataStructure();
5                 // [...]
6                 GUIMain.setDataStructure(vt);
7         }
8         // [...]
9 }
```

Before we continue to describe the single components, we want to mention briefly, that all calls are performed in accordance with the guidelines of the Java API. That means that a particular method, which starts with a "get-", reads the data structure, whereas, the method, which starts with a "set-", writes in the data structure in a controlled way, as long as writing is allowed. To give an example, following the above code segment4.1, the call for writing in the data structure would be "setDataStructure", as presented in line 5. For the sake of simplicity, here we will only declare the call with "get".

4.2.1 Data Structure Hierarchy

In the next subsections, the single data structure elements, as presented in table 4.2 will be explained in detail. For a better understanding of the next items, in figure 4.1 a class diagram is illustrated, which demonstrates, with which method calls and return values the GUIMain class offers its data. In the middle right, the GUIMain class is pictured, which administrates all data at a central point.

Get the DataStructure

The main data structure of the NS-Mapper can be accessed by the getDataStructure call, which returns a ValueTable object, lookup 4.1. This object integrates the information about the scenario and provides the lists of the mobile Nodes and MovementField. The following methods can be called from the ValueTable object:

getSzenarioOptions() This method returns a ScenarioOptions object, which contains organizational values, such as the name of the scenario,

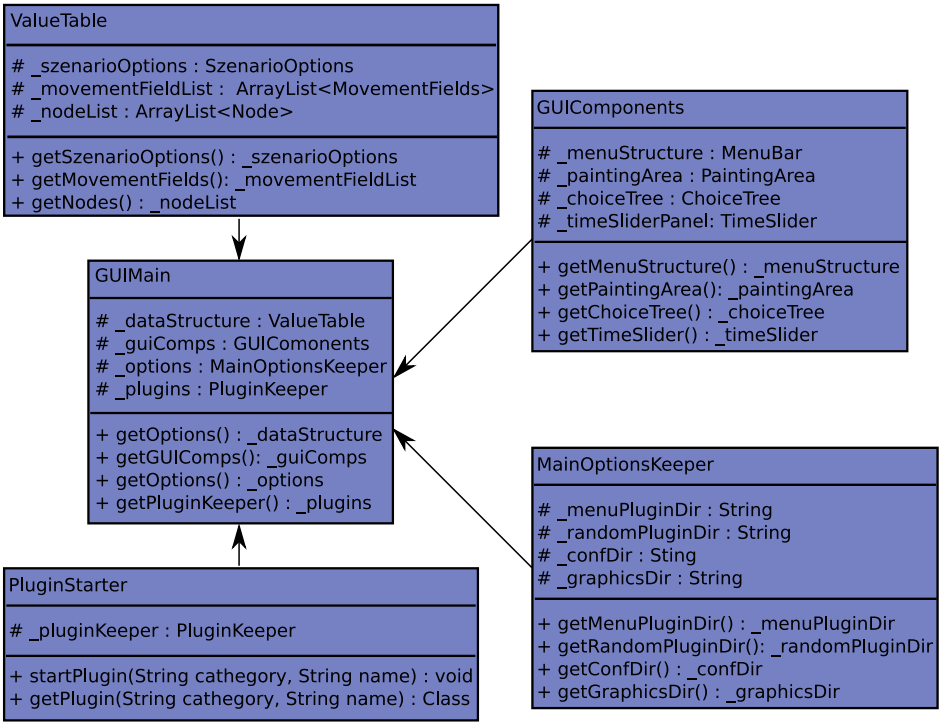

Figure 4.1: Data Structure Class-Diagramm

the dimension, the whole simulation time and the used image for the background, presenting a map or a ground plan.

getMovementFields() This method returns an ArrayList with all MovementField objects, which contains its position or its paths, and whether it is a dynamic MovementField, which can move through the scenario.

getNodes() This method returns an ArrayList of mobile nodes, containing information about their positions at corresponding time points in form of a TimeScheduler, which administrates and stores all paths in the NS-Mapper.

Get the GUI Components

The GUI components of the NS-Mapper can be reached by the getGUIComps method and return a GUIComponents object; figure 4.1 shows its class. It keeps the references to all GUI-Elements. Figure 4.2 represents the NS-Mapper and marks different elements, returned with the following methods:

1. **getMenuBar()** This method returns a Java MenuBar GUI element and keeps and coordinates the menu structure of the NS-Mapper.

2. **getPaintingArea()** This method returns a PaintingArea object, which controls the visualization of the scenario and is also used for modelling the MovementFields, mobile nodes and their paths. Therefore, the PaintingArea is an important part in the design of the plugins. It will be described in the subsequent section.

3. **getChoiceTree()** This method returns a Java ChoiceTree object, which represents a virtual data structure tree and shows information about the scenario, the MovementFields and mobile nodes.

4. **getTimeSlider()** This method returns an Java TimeSlider object and controls and changes the timing of the NS-Mapper. This object is especially important for the modelling of paths of MovementFields and mobile nodes, because it can reproduce the course of the scenario in a time context.

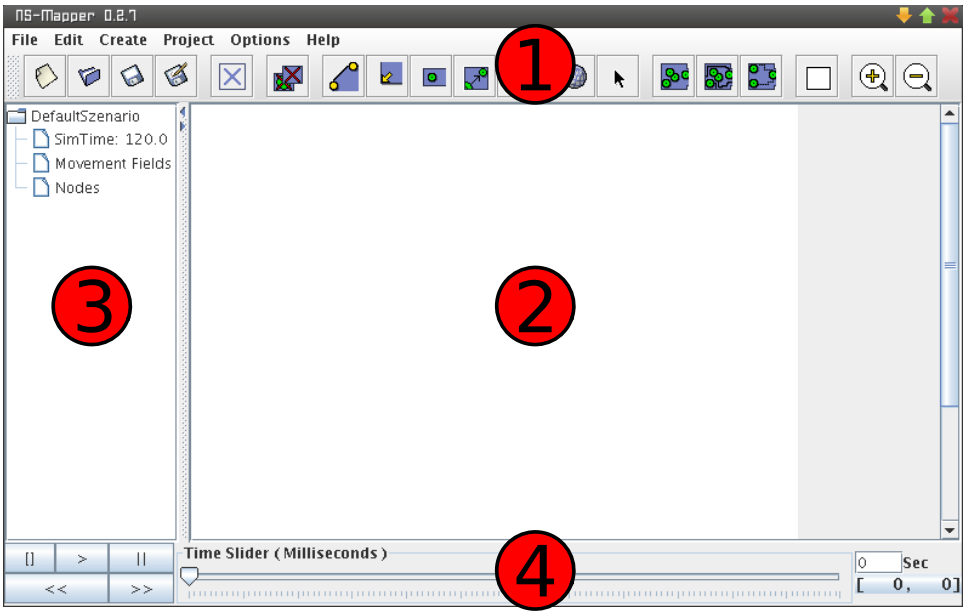

Figure 4.2: GUI Components in the NS-Mapper

Get the Options

The getOptions method returns the MainOptionsKeeper, which stores additional information about the NS-Mapper itself, such as paths to the configuration folder, icons and plugin directories, as well as temporary values, such as version, the current file name of the scenario and further options needed by the PaintingArea, such as the colour of all painted elements and other GUI components, and finally, temporary options of the node and MovementField data structure. The class diagram 4.1 presents only the most important methods and return values of the MainOptionsKeeper.

Start Plugins

The PluginStarter, previously explained in section 3.4.2, returns the PluginStarter class, which coordinates the loading, storing and the execution of plugins, lookup class-diagram 4.1. The PluginStarter class contains the following method calls:

startPlugin(String category, String name) This method starts a special plugin. Its plugin category and the name of the category are sent to a method by means of a string. The following categories can be consigned by the string: “startPoint”, “movement”, “traffic” und “menu”. Whereas “startPoint” refers to the start point plugin, etc. The string, which transfers the name to the startPlugin method, depends on the names defined by the plugin developer.

getPlugin(String category, String name) The PluginStarter has to be capable of returning particular plugins to enable plugins to exchange data and optionally, execute program parts. This is capsulated in a class object.

4.3 NS-Mapper Framework

The framework is an important part of the NS-Mapper. It was developed in this diploma thesis, to easily implement plugins. The framework consists partly of classes, which are integrated in the upper data structure, distributed by their tasks, and partly of autonomous classes, which can be imported by the plugins. It guarantees the reusability of calculations in the program, standardizes interactions and simplifies recurrent tasks, as for

example the PathLine class, which controls the generated courses of motion of mobile nodes in a plugin, and if necessary, corrects or adjusts them. Furthermore, it controls the preparation and exchange of data and prevents a direct manipulation of them, as well as a saving with incorrect values, as for example the OptionsKeeper class, which guarantees a standardized exchange of data between the plugins. Additionally, the framework makes data structures available, which provide the plugin developer with information and calculations about the status of the main program. It controls the access from Plugins to GUI components and increases their functionality. It also translates, transfers and simplifies the calculations and the data from and to plugins, to adjust them to the limited capabilities of the NS-Mapper core program. This framework can be accessed with the above listed methods in section 4.2.
In the next subsections, the most important tasks, which were developed in this thesis, are described. They are necessary for the development of random based plugins. It is explained how to use the framework. Furthermore, we will describe how the NS-Mapper controls the general access to information and objects, such as the painting of intermediate steps or the access of marked elements. Additionally, we describe how to process interactions between a plugin, the core program and the user by keyboard and mouse events. We will describe the implementations of the OptionsKeeper and the PathLine class, which were previously defined in section 3.5.2 “Random Options” and section 3.5.4 “NS-Mapper relations”.

4.3.1 NSMCleaner

At first, we have to clarify the usage of the NSMCleaner class, because it is used in almost all code examples during the explanation of the NS-Mapper framework. It is used for deleting or setting the options to default.
The NSCleaner is accessible by the GUIMain.getOptions().getGUIOptions() .getCleaner() call. It is a small class, which deletes data structure or event based processes. Some features are important for the work of randomized plugins in the NS-Mapper. The relevant features are now described briefly and will only be explained in detail, if they are used in connection with the random tasks. This class contains the following methods:

setMouseAnnouncerToDefault()	Deletes all mouse events of a plugin from the NS-Mapper.
setKeyboardAnnouncerToDefault	Deletes all keyboard events of a plugin from the NS-Mapper.
cleanNodePathsAll()	Cleans all movement paths of all wireless nodes, which are made by a random mobility model.
cleanNodePaths(String)	Cleans a specific path, which was made by a specific random mobility model.
cleanConnections	Deletes all connections between wireless nodes, which are made by a random traffic model.

4.3.2 PaintingArea

The PaintingArea class contains the most important images, such as MovementFields, wireless nodes and optionally, paths to the simulation environment. But sometimes it is necessary to draw more complex figures and to gain more information of intermediate steps of this PaintingArea. Therefore, the plugins need direct access to that class, which also allows them to paint in the NS-Mapper. For example, if a plugin generates automatically a grid for limited moving, e.g. for a graph based mobility model, the user must have knowledge of what is represented by the calculated meshes to evaluate their level of detail. Now he can decide, depending on the visible graph, which configurations or levels fit best. That makes the handling of plugins easier and provides the person, who uses a plugin, with more transparency regarding the view of the course of events.
The different figures, which can be handled with the PaintingArea are stored in the objects for drawing: RectangleCreation, MovementFieldCreation, MovementFieldMarked, MovementFieldEdgeMark, NodeMarked, PathLine and Line and also an ArrayList to combine the drawing of more than one element, which can be: Line, PathLine and DrawString:

RectangleCreation	Draws a black rectangle.
MovementFieldCreation	Draws a red rectangle with separately circled edges.
MovementFieldMarked	Draws a normal polygon.
MovementFieldEdgeMark	Draws a black polygon with separately circled edges.
NodeMarked	Draws a black node. If the marked value is true, the node will be drawn red.
PathLine	Draws a not-closed line path with separated circles at every line edge.
Line	Draws a black line.
DrawString	Draws a black text, with or without a dot at the String position.

Table 4.3.2: Graphical Elements for the PaintingArea

When a plugin draws its elements in the PaintingArea, as presented in table 4.3.2, they are put on the top of all existing drawings of the modelling level of the scenarios, as for example, the visualised MovementFields, the mobile nodes and their paths, because the focus is always put on the active plugin, and less on the graphic presentation of the scenario. More information about these object, their constructors and their class methods can be found in the NS-Mapper JavaDocs.
The usage of these objects is very easy and can be added directly to the data base of the PaintingArea with the setInCreation method of the static GUIMain call getElementInCreation (lookup section 4.2). Listing 4.2 provides an example, which draws a text to the PaintingArea with a dot at the String position. After the instantiation of the ArrayList, the new DrawString object declares a text, followed by its position and a boolean value, which additionally declares to paint a dot at the given position, line 1 and 2. The ArrayList with the added DrawString object is given to the setInCreation method, a static reference in the GUIMain, line 3. At the end of the code fragment the drawing is deleted again by setting the database to a **null** object in line 7 or by calling the NSMCleaner from line 8.

Listing 4.2: Drawing a String to the PaintingArea

```
1 ArrayList al = new ArrayList();
2 al.add(new DrawString(``Example String'', new Position(10, 10),
      true));
```

```
3 GUIMain.getOptions().getGUIOptions().getElementInCreation().
      setInCreation(al);
4 // [...]
5 GUIMain.getOptions().getGUIOptions().getElementInCreation().
      setInCreation(null);
6 // ... or ...
7 NSMCleaner cleaner = GUIMain.getOptions().getGUIOptions().
      getCleaner();
8 cleaner.setInCreationToDefault() {
```

The deletion of the String object indicates that this data base is only available temporarily, until the same or another plugin overwrites these definitions. Additionally, it does not interfere with the level of paintings, which are produced by the simulation area, containing the MovementFields, wireless nodes and paths. The two levels are independent of each other. Therefore, the developer of a plugin can manipulate his paintings, without affecting the simulation based representation.

4.3.3 Handle mouse events

To let a plugin interact with a user by mouse support, the plugin does not need to extend the abstract class RandomFrame (lookup Listing 4.3), but extends the MouseAnnouncementFrame class, which itself extends the RandomFrame in line 3. It gives the plugin the ability to link its functions to a predefined method into the MouseListener of the GUIMain class. At first, the developer decides which mouse feature he wants to use. The possibilities are stored in the MouseEnum class and are: CLICKED, PRESSED, DRAGGED, RELEASED and MOVED. In line 10 of the code fragment an array of enum types is declared, which the developer wants to use. The next line calls the predefined announce method of the extended MouseAnnouncementFrame class containing this array. It informs the GUIMain class that a plugin is called for a specific mouse function. There are two different ways to inform the GUIMain to link functions with the MouseListener: Either with the already described enum array in line 10 and its announcement in line 11, or directly with an enum type in line 13, collected one by one.
Now the developer has to declare the predefined methods, he announced, starting from line 11. The names of the entrance points and the values are the same as declared from the MouseListener made by the Java API: mouseClicked, mousePressed and so on, like the MouseEnum class. The MouseEvent objects are the values, which are leading to these methods.

They are created by the Java MouseListener containing information, such as the mouse position or which button has to be pressed. The only difference to the real MouseListener in Java is that these classes return boolean values. A **true** value processes all further mouse events, not the events of a plugin, but the events of the NS-Mapper core program, whereas a **false** value immediately exits all mouse processes.
In the method mouseClicked from line 18 the code fragment deletes its mouse interruption from the GUIMain, if it knows that this service is already executed. It can be done by calling a cleaner object with the setMouseAnnouncerToDefault method.

Listing 4.3: Use of the mouse announcement

```
1  import mapper.dataStructure.announcer.MouseAnnouncementMFrame;
2
3  public class ExamplePlugin extends MouseAnnouncementMFrame {
4          public boolean init() {
5                  NSMCleaner cleaner = GUIMain.getOptions().
                       getGUIOptions().getCleaner();
6                  cleaner.setMouseAnnouncerToDefault();
7                  // [...]
8          }
9          public boolean run() {
10                 MouseEnum mouseEvents[] = { MouseEnum.CLICKED,
                       MouseEnum.MOVED };
11                 announce(mouseEvents);
12                 // ... or use ...
13                 announce(MouseEnum.DRAGGED);
14                 // [...]
15         }
16
17         public boolean mouseClicked(MouseEvent event) {
18                 // [...]
19                 NSMCleaner cleaner = GUIMain.getOptions().
                       getGUIOptions().getCleaner();
20                 cleaner.setMouseAnnouncerToDefault();
21         }
22
23         public boolean mouseMoved(MouseEvent event) {/*[...]*/}
24
25         public boolean mouseDragged(MouseEvent event) {/*[...]
               */}
26 }
```

Nevertheless, we have to note a specific feature of the mouse announcer, which might create some confusion. If a developer implements a plugin with

mouse events, it is usually best, if he cleans all previous mouse announcements from the GUIMain at the beginning of the execution. For example, as in the init method of the above listing in line 6, because there is regularly only one activ Plugin at a time. It is imaginable that the calculations of a previous plugin is not finished regularly if, for example, the user has executed a plugin and changes his mind by immediately executing another plugin. In this case some announced mouse events can still point to the previous plugin, even if the new plugin is executed. This behaviour is not a weakness, because the PluginStarter can clean theoretically all events of the NS-Mapper at the beginning of every plugin execution. It is a wanted feature, because this way it can support the interactions between two different plugins, by not pre-deleting all mouse events.

4.3.4 Handle keyboard events

To enable key support for a plugin, the developer has to import the mapper.dataStructure.announcer.KeyboardAnnouncement and to implement this interface, see line 1 and 3 of the code fragment 4.4. The announcement of keys is as easy as the announcement of mouse events. The execution of all of them is done by a single method, because the huge quantity of keys makes the individual handling of every single key too uncomfortable. That is why the run method in line 6 calls a global announcement of keys and processes it in the KeyProcessor method, as in line 8. As for the mouse events, the returned boolean value for the KeyProcessor method is used to inform if the plugin allows the core program of the NS-Mapper to execute its keyevents. If the boolean value is **false**, the core program is not allowed to execute keyevents; if the boolean value is different, it is allowed. This method inquires the data of the KeyListener by invoking the GUIMain.getOptions().getGUIOptions().getKeyListens().isKey(String) method by asking for a key. The key is reached by a String and returns a boolean value, when the key is pressed, see line 9.
Like the mouse announcement, the process of key events can be deleted by the NSMCleaner class, if the work is done and no more key events, which use the setKeyboardAnnouncerToDefault method, have to be processed.

Listing 4.4: Use of the keyboard announcement

```
1 import mapper.dataStructure.announcer.KeyboardAnnouncement;
2
```

```
3 public class ExamplePlugin implements KeyboardAnnouncement{
4         public boolean run(){
5                 // [...]
6                 KeyAnnouncement();
7         }
8         public boolean KeyProcessor() {
9                 boolean key = GUIMain.getOptions().
                      getGUIOptions().getKeyListens().isKey("STRG
                      ");
10                // [...]
11                NSMCleaner cleaner = GUIMain.getOptions().
                      getGUIOptions().getCleaner();
12                cleaner.setKeyboardAnnouncerToDefault();
13        }
14 }
```

4.3.5 Access marked elements

The NS-Mapper allows the identification and the access to elements, which are marked in the PaintingArea, for every plugin component. This access is at the moment limited to nodes and MovementFields, exactly all those components, which are important in the PaintingArea. The marking of paths of mobile nodes and dynamic MovementFields is supposed to follow in the near future.

If the user wants to mark more than one element, he can group them by pressing the control key. Listing 4.5 shows how to reach an element from the marker class, which collects the marked elements. All elements are stored in an ArrayList object, which can be accessed from the GUIMain with a GUIMain.getOptions().getGUIOptions().getMarked().getElements() call. The ArrayList stores all single objects as elements, which are a basic representation of nodes and MovementFields. Thus, every single object must be casted from an element object to their main object. E.g., the polygon edges of a MovementField belong to the category of nodes, which therefore, have to be accessed by this category. This element is not handled separately.

Listing 4.5: How to get a marked Element

```
1 ArrayList al = (ArrayList) GUIMain.getOptions().getGUIOptions()
      .getMarked().getElements();
2 for(Object element : al) {
3         if(element instanceof Node) {/*[...]*/}
4         if(element instanceof MovementField {/*[...]*/}
```

```
5 }
```

4.3.6 OptionsKeeper

As described in section "Random Options" 3.5.2, the OptionsKeeper class standardizes the declaration and the access to variables, as well as their exchange within and between plugins. It also generates an automatic GUI for their manipulation. The whole range of functions, made or influenced by the OptionsKeeper, spreads across three hierarchical processing levels, whereas the OptionsKeeper class itself is located in the middle.
The OptionsKeeper uses a container class for declaring and storing all various variables, for guaranteeing a standardized declaration and access to different rudimentary data types, such as integer, double, boolean, etc. This container class is called RPluginOptions and represents the lowest level of this hierarchy. The standardization of this container allows the OptionsKeeper to hide the complexity of administrating and handling the different values from the plugins and the developer. At the middle level, the OptionsKeeper stores and administers all values and combines - depending on the RPluginsOptions - a panel with the appropriate GUI elements for their editing. The highest level of the OptionsKeeper processing is represented by the RandomFrame, previously described in section 3.5.1. The RandomFrame coordinates the preprocessing of values made with the OptionsKeeper and creates a readyto use manipulation dialog.
In the following, we describe how the OptionsKeeper simplifies the use of options in random plugins.

Unified Data Declaration

As briefly described above, the RPluginOptions is a container class for a bunch of different data types. It standardizes the handling of rudimentary data structures and extends them with additional information, such as a short categorisation of the value and a physical unit. This information allows to build an automated graphical user interface for the given values, see figure 4.3. The following three constructors are supported:

```
public RPluginOptions(String info, String unit)
public RPluginOptions(String info, String unit, TYPE value)
```

```
public RPluginOptions(String info, String unit, ARRAY values,
    TYPE defaultValue)
```

The info String represents the exact categorization, which gives special information about the value, for example: “Min Speed” or “Max Speed”. The unit String represents the physical unit of the value, like “km/h” or “meter”. The value TYPE represents a value out of a wide range of rudimentary data structures, such as byte, short, int, long, float, double, char, String, boolean. And finally, the ARRAY values represent a whole array of the same data structures as used in TYPE.
The three constructors guarantee that the declaration of the data types is reduced to the most important common denominator. The first constructor creates only an object, which functions as a separator in the GUI processing. It keeps the declaration of the info and unit String only for tagging. The second constructor defines additionally a TYPE, representing only one single value of a given option. This value is completely editable and its GUI representation is a JTextField from the Java API. Only if it is a boolean value, the value is represented by a JCheckBox. The third constructor has an additional array of values, which are from the same data structure as their TYPE. The array enables the options dialog to define a limited range of values, which are valid in the options context. In this case the TYPE value at the end of this constructor represents the default value, which also must be listed in the array. The GUI representation is a JSpinner or a JComboBox, depending on the size or the array.
All values, stored in a RPluginOptions object, remain type pure and are not casted to another rudimentary data structure. Therefore, no faults or changed values can appear by using this container class.

Declaration of OptionsKeeper Values

When a developer implements a random plugin, he can define commonly used values with the makeDefaultOptions method. This declaration is a central aspect between a random plugin and its interprocess communication within the NS-Mapper. The makeDefaultOptions method is firstly declared in the RandomFrame interface in section 3.5 and is intended for a regular instantiation of all commonly used values in the plugin, similar to a constructor. When a random plugin is started for the first time, this method will always be called from the PluginStarter, shortly after its loading into

the memory.
The Optionskeeper class facilitates the work of the developer. This class is explained in section 3.5.2 and should be the preferred selection for the introduction of values, which are necessary for the random algorithm itself or for the manipulation process between the user and the plugin. It supports the developer in all steps, which are important for the process, from starting the declaration, across the manipulation, till the distribution and interaction with other plugins.
Listing 4.6 shows a common processing of the makeDefaultOptions method by using the OptionsKeeper. The general name of a OptionsKeeper object uses the suffix "_options" and is declared in the abstract class "RandomFrame". In line number 5 the _options object is cleared from all its values first, in case former changes of the values of the random plugin are still stored in this object. If a user creates some wrong configurations, which are not intercepted by the OptionsKeeper class, it can re-change all values to the clean, original status by calling the makeDefaultOptions method again. The reason why the OptionsKeeper does not always intercept all wrong values in the manipulation process, is described in section 3.5.6.
From line number 7 to 15 in the listing, the code fragment declares common variables for the OptionsKeeper. For example an integer array in line 8 and 9 or a single double value in line 15. The declared variables are inserted into the "_options" object with the method name "add", which is invoked by a RPluginOptions object.

Listing 4.6: Setup the default options

```
1  public class RandomAlgorithm extends RandomFrame {
2
3          @Overwritten
4          public void makeDefaultOptions() {
5                  _options.clear();
6
7                  // OptionsKeeper values
8                  int pedestrian[] = {4, 10};
9                  _options.add(new RPluginOptions("Pedestrian␣
                       Speed:", "km/h", pedestrian, pedestrian[1])
                       );
10                 int vehicle[] = {20, 50, 60, 100, 120};
11                 _options.add(new RPluginOptions("Vehicle␣Speed:
                       ", "km/h", vehicle, vehicle[1]));
12                 String str[] = {"NORMAL", "GAUSS"};
```

```
13                  _options.add(new RPluginOptions("Random␣Speed␣
                        Function:␣", "", str, "GAUSS"));
14                  _options.add(new RPluginOptions("Random␣Time:",
                         "", false));
15                  _options.add(new RPluginOptions("StartTime:", "
                        ms", 12d));
16                  _options.add(new RPluginOptions("StartTime:", "
                        ms", 65d));
17
18                  // other global values
19                  _tmp = 0;
20                  pos = new Position(0,0);
21          }
22          [...]
23  }
```

Starting from line number 18, there is an example of declared variables, which might be necessary for the processing of the plugin. But as they are not important for the manipulation and interaction process between the random plugin and the user, the values are not registrated in the OptionsKeeper.

Manipulate Values with the OptionsKeeper

By means of the OptionsKeeper method "getPanel", the GUI dialog can be called. This method returns a plain JPanel with all values specified in the makeDefaultOptions method, without any additional control buttons, such as an "OK" or "Cancel" button. So this plain panel can be integrated into every GUI element declared in the Java API. Mainly the getOptionDialog method in the abstract class RandomFrame prepares this GUI automatically for the developer. Unless the developer does not overwrite this method, he gets the dialog from the RandomFrame predefined from the OptionsKeeper, which is ready to be used. For example, the figure 4.3 presents the GUI, made by the declaration of Listing 4.6. If the developer wants to have additional elements in this dialog or if he wants to extend the functionalities of the OptionKeeper, he has to overwrite the getOptionDialog method in the RandomFrame.

Return and Process Values with the OptionsKeeper

To return values from the OptionsKeeper, the developer has to call the get method. This method is invoked by an integer value, which represents the

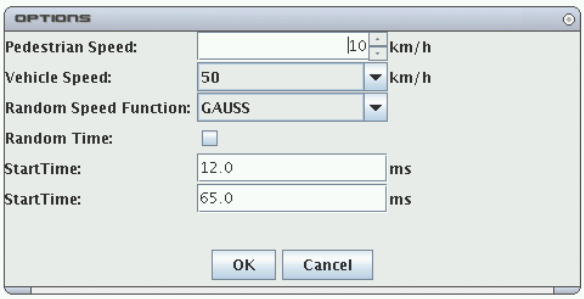

Figure 4.3: Options Dialog

position of the value in the OptionsKeeper list. These values are in the order of their declaration, made by the makeDefaultOptions method in the RandomFrame class.

Listing 4.7 sets an example how the single values of the OptionsKeeper can be accessed. A code fragment like this might appear in a run method of a random plugin.

In line number 3, the different values are called by a "for" loop, differing by the incremental index i. In line 4 and 5, the name and the unit values are returned with the getInfo and getUnit method. With these values the developer can specify the way, in which a value has to be processed, like in line number 10. The switch statement in line number 4 reallocates the values to their rudimentary data type. The getDataType method returns the enum values of the data type at the specified position of the index.

The DataType enum categorizes the stored values of the OptionsKeeper, so that their true data type can be recognized, like in line number 8 and 9.

The enum class DataType declares the following values: NONE, BYTE, SHORT, INT, LONG, FLOAT, DOUBLE, BOOLEAN, CHAR, STRING, OBJECT, BYTE_ARRAY, SHORT_ARRAY, INT_ARRAY, LONG_ARRAY, FLOAT_ARRAY, DOUBLE_ARRAY, CHAR_ARRAY, STRING_ARRAY, ERROR. The NONE enum declares a value, which is not reachable (for more detail see: the constructors for the RPluginOptions class in section 4.3.6), and an ERROR enum, if a developer wants to call an incorrect value.

Listing 4.7: Return and processing of values

```
1 String name, unit;
2 byte value;
3 for(int i = 0; i<_options.size(); i++) {
```

```
4           name = _options.get(i).getInfo();
5           unit = _options.get(i).getUnit();
6
7           switch(_options.get(i).getDataType()) {
8                   case DataTypes.BYTE:
9                           value = _options.get(i).getByte();
10                          if(name.equals("Maximal Speed"))
11                                  //[...]
12                          break;
13                  case DataTypes.INT;
14          [...]
15          }
16  }
```

Access Options of a Plugin

Each random plugin in the NS-Mapper has its own OptionsKeeper object, see 3.5.1. The plugin can use it to administrate all data, which is necessary for an interaction of the user with the settings of the plugin. The OptionsKeeper object of the plugin can be described as the master object. All changes of the data, performed by the user, are saved in here. The OptionKeeper of a plugin can be reached by the GUIMain:

```
String cathegory = "movement";
String name = "GaussMarkov";
OptionsKeeper oK = ((RandomFrame) GUIMain.getPluginStarter().
    getPlugin(cathegory, name)).getOptions();
```

As presented in the example listing above, the OptionsKeeper of the "Gauss Markov" mobility model is returned. For that, in the PluginStarter the method getPlugin is called, which demands two String parameter, the category and its name. By the first String, the plugin category is defined, whereas the following strings are allowed: "startPoint", "movement" und "traffic". The second string refers to the plugin itself and states its name. The names are defined in the configuration file, see section 4.6.1. The plugin object, which is returned by the getRandomPlugin method, must be casted to a RandomFrame. The interface, RandomFrame, allows access to the getOptions method, which returns the OptionsKeeper object of the random plugin, which is stored in the oK value.
When a special random plugin is appointed to a mobile node in its declaration dialog, e.g. to a mobility motion plugin, a copy of its OptionsKeeper object is constructed and written in the mobility model option of the mo-

bile node, together with the name of the plugin, see 4.6.2. This action is performed automatically, thus, the plugin developer does not have to deal with it.
Every mobile node can maximally contain three different OptionsKeepter objects, as long as it is appointed to a random plugin in the declaration dialog: one OptionsKeeper for a random start point plugin, one for its motion and one for its traffic plugin. But not every node gets its own options representation, because the user can define a group of nodes, which share their option objects with each other. Therefore, if a local representation of an OptionsKeeper of a single node is changed, every single node of the same group with the same OptionsKeeper reference is also changed. The options of a node are returned by the GUIMain with the following statement:

```
Node node = GUIMain.getDataStructure().getNodeList().get(id);
OptionsKeeper oK = node.getRandomNodeOptions().get(cathegory);
```

The “id” value assigns the node to its position in the node list, indem alle mobilen Knoten des NS-Mappers gespeichert sind. The “category” value for reaching the OptionsKeeper is the same as explained above: “startPoint”, “movement” or “traffic”.

4.4 Nodes and MovementFields

Also an important part of the plugin developement is the access of Nodes and MovementFields. All mobile Nodes and MovementFields objects can be accessed by the GUIMain, whereat these objects are stored in an ArrayList, which needs an index, to choose a specific one:

```
int id = 3; // The index of a Node or MovementField
Node node = GUIMain.getDataStructure().getNodeList().get(id);
MovementField mf = GUIMain.getDataStructure().
    getMovementFieldList().get(id);
// [...]
```

A mobile node is a specific point, with x and y value for its position and MovementFields, which is exactly a polygon, keeps a list of this positions. The current position in the timeline of their movements is reached by:

```
// [...]
int time = 55;
Position currentNodePos = node.getCurrentPosition(time);
Polygon currentMovementFieldPos = mf.getCurrentPolygon(time);
```

But this Positions of a node and a MovementField object is really stored in a TimeScheduler object, which is declared in every of these two elements. The TimeScheduler is briefly described in the next section.

4.5 TimeScheduler

Before explaining the PathLine class and its usage, at first we want to mention briefly the TimeScheduler. Because it was already described in detail in the study thesis [8], here we only want to characterize its behaviour.
Every dynamical element in the NS-Mapper, precisely the MovemenFields and the mobile nodes, possesses a TimeScheduler, which contains information where an element is located at what time. In principle, the TimeScheduler represents a list of discrete positions, which are related to a discrete time point. Because in the NS-Mapper only lineal movements exist, from position a to position b containing a start time t_1 and a stop time t_2, it can be precisely calculated, where an element is located at a special time point. This TimeScheduler is used in the following PathLine class.

4.5.1 PathLine

The PathLine class, see section, was developed for the geometrical calculation of paths in 2D. It was specially adjusted to the needs and restrictions of the NS-Mapper. It calculates the way of an element, for example, of a mobile node from a position a to a position b, in dependence of the time, from a start time t_1 till a stop time t_2.
Depending on the kind of dynamical element, which invokes the PathLine, it is decided how the calculations are performed. If it is a MovementField, the PathLine class only has to put the path in the TimeScheduler, because this object does not possess any restrictions concerning its movement. But it is different if it is a mobile node, whose movements are restricted to a MovementField, which can be only left when immediately passing to another touching MovementField. At no time, the node is allowed to take a position outside a MovementField. Because in the scenario dynamic MovementFields may exist, which also move, every single movement of a mobile node has to be controlled at each time point, and if necessary its movement has to be corrected or forbidden. It is the PathLine class, which performs these calculations, as already described in 3.5.4.

The constructor of the PathLine in the following code part declares an object element, which represents the node or the MovementField, which has to be calculated. The second variable of the constructor is the TimeScheduler object.

```
public PathLine(Object element, TimeScheduler timeScheduler)
```

The easiest way to explain the structure of PathLine and the usage of random plugins is to cite an example of a code fragment, as presented in listing 4.8. The fragment symbolizes the processing of how the calculations of a random plugin can be executed by a single mobile node. In line number 2, the PathLine object is instantiated with the given node object and a copy of its existing TimeScheduler. The use of the copy method of the TimeScheduler is very important, because this way the node returns a reference to its objects, and we do not want to write calculations directly into the real TimeScheduler of the node. The node then returns a reference of its object. In line number 3, we define the start time of the calculation, by appointing it a stop time of its last calculation step. If no previous calculation step exists, the previous stop time is 0. This procedure is adjusted to the fact that the TimeScheduler might have some unfinished paths, because the PathLine class is not only used by random plugins. But random plugins start in general with an empty TimeScheduler, so the start time is usually zero. In line number 4, we get the last TimeSlot from the TimeScheduler. This object contains information such as start and stop time, as well as a start and a stop position, which are necessary for the line calculations, previously described in section 3.5.4. The last TimeSlot with its stop values for time and position forms the basis for the start time and position of the new TimeSlot, given to the PathLine class in line number 10. In line number 6, we have a loop, which iterates till the end of the simulation time. Beginning from line 7, the random process of the plugin can be started. The random plugin can calculate a new TimeSlot, which has a new random stop time and position. In line number 10, the PathLine gets the new TimeSlot object and decides if the whole TimeSlot has a regular movement. It is the same behaviour, we explained previously in the design concept of the class in section 3.5.4. Shortly and simplified, it lets the moving node pass across a MovementField border if another MovementField can catch the node. If not, it is repelled at the border. This line is surrounded by an exception handling: The TimeSlot can throw a SimulationTimeException, if the TimeSlot

contains a wrong value, e.g. if the start time is higher than the stop time, or if the stop time is higher than the simulation time, etc. In line number 12, the new stop time is returned to the time value for breaking the loop. If the loop is finished and the TimeScheduler contains correct values, this is transferred to the present node and the next node can be calculated (line 15).

Listing 4.8: The usage of PathLine

```
1  node = ... // A wireless node
2  PathLine pl = new PathLine(node, node.getTimeScheduler().copy()
       );
3  double time = node.getTimeScheduler().getLastSlot().getStopTime
       ();
4  TimeSlot ts = node.getTimeScheduler().getLastSlot();
5
6  while (time != GUIMain.getDataStructure().getValueNew().
       getSimulationTime()) {
7          // generate a new random TimeSlot ....
8          // [...]
9          try {
10                 pl.setSlot(ts);
11         } catch (Exception e) { e.printStackTrace(); }
12         time = node.getTimeScheduler().getLastSlot().
               getStopTime();
13 }
14 if(timeSchedulerIsCorrect(pl.getTimeScheduler())
15         node.setTimeScheduler(pl.getTimeScheduler());
```

Finally we have to note that the PathLine is only responsible for collecting and sorting those intersections and calculations, which result in a crossing of the MovementField border. The real intersection and distance calculation are done in the mapper.dataStructure.elements.line.Line class.

4.6 Random Plugin Declaration

In this section, we will explain, how to integrate a plugin in the NS-Mapper.

4.6.1 Integrating a Random Plugin

In contrast to the past NS-Mapper versions, now all plugins are not automatically found and loaded into the main programme, as described previously. Now the representation of the random plugins is edited in the "ralgorithms.conf" file in the conf directory. The benefit of this proceeding

is to only load a plugin into the memory, if needed.
The path of the random plugin directories is "CLASSPATH/mapper/plugins/randomAlgorithms". In this path the classes are divided into three random sections and are stored in the subdirectories "movementPlugins", "startPointPlugins" and "trafficPlugins". To conclude, it is only necessary to edit the plugin name to the "ralgorithms.conf" and they are integrated. This is shown in Listing 4.9.

Listing 4.9: ralgorithms.conf

```
# The ralgorithms.conf is subdevidet in the three
# random plugin cathegories: "Random Start Point",
# "Random Movement" and "Radnom Traffic". New plugins
# can edited in these Sections. The Java class files
# have to put into the ".../mapper/plugins/
# randomAlgorithms" plugin directorys. You can
# define subdirectories by specify it with the
# full path without class extension. Example:
# "..." {
#       "<subdirectory>.<subdirectory>.<randomPlugin>"
# }
"Random Start Point"{
        "BoundingBoxRandom",
        "MotionBasedPreCalculation"
}
"Random Movement"{
        "RandomWayPoint",
        "GaussMarkov"
}
"Random Traffic"{
        "RudimentaryTopology",
        "BitTorrentTopology"
}
```

4.6.2 Declaration of Plugins

The execution of a particular random plugin is started by invoking the "Add Node" dialog, which appears when someone wants to declare a mobile node, see picture4.4. Point 1 of the picture marks the random based categories with "Start Point", "Movement" und "Traffic". The checkbox declares the category, which is valid and will therefore be executed. If, for example, the checkbox declares "none", no random based plugin is loaded for this particular point. As long as the buttons of point 2 or 3 are not pushed, the particular plugins are only loaded when demanded for execution. Point 2

marks the options, offered by the makeDefaultOptions method, which can be edited to change the behaviour of the mobile nodes. Point 3 opens the Help dialog, which is contained in the Plugin Interface. This button opens a dialog, which explains special option settings, as long as they are not self-explanatory.

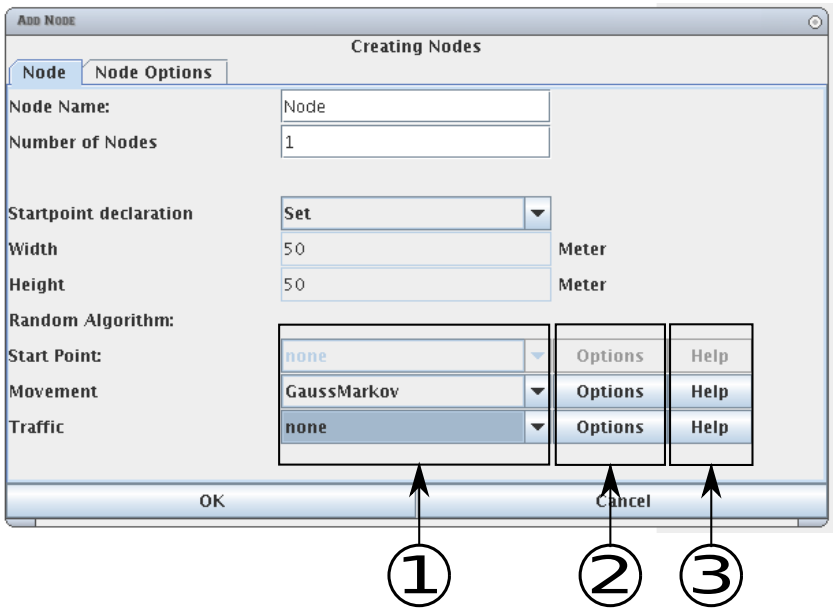

Figure 4.4: Add Node Dialog

4.7 Implementation of the Random Plugins

In this section, we will explain the implementation of the random based Plugin, without describing again, how a single, selected plugin works. This was already done in chapter 2.3. Now we will explain the adjustments, which were necessary to adapt the plugins to the particular capabilities of the NS-Mapper. Partially we had to strongly expand the plugins to fit this demand, ensuring that the calculations can be carried out correctly.

4.7.1 Random Start Point

In this section the "Mobility Motion Based Pre-Calculation" plugin for the category start point is explained. At first we will explain which values are

necessary for its regular execution, and then, we will describe how it works.

Available Options

The modus operandi of the "Mobility Motion Based Pre-Calculation" plugin was already explained in section 2.3.1 und 2.4.1. In its implemented form it offers the following options, which can be defined by the user (see figure 4.5, representing the dialog of the plugin options):
The first option defines the duration of the pre-calculation. It decides the time period, in which the wireless nodes move preliminarily. Their end position after this pre-calculation will form their new starting points for the subsequent process. The unit is declared in seconds.
Afterwards, the user can choose the mobility model from a list of available plugins. This option is called "Motion Algorithm". It leads to a button, which contains the plugin settings, because every mobility model plugin possesses its own settings.
Finally, the option "Start Point Algorithm" will offer a list of available node placement plugins. At the beginning the nodes do not possess a specified placement in the simulation area. That is why, at first, a temporal, random placement of the nodes occurs for the pre-simulation.

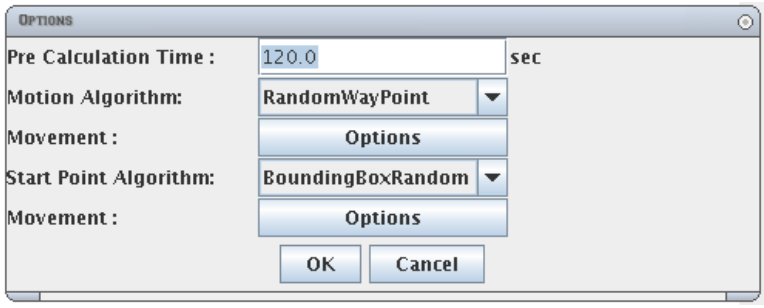

Figure 4.5: Motion Based Pre Calculation Options Dialog

Implementational Aspects

Striktly seen, the "Motion Based Pre Calculation" plugin practically does not calculate a start point itself, but it only manipulates already existing values and coordinates the course of calculation, by consigning single mobile nodes to a different random mobility motion plugin, which is responsible for

the movement of the node. If the path calculations are concluded, the "Motion Based Pre Calculation" plugin inserts the last stop position of a node and deletes its path.
For making the "Motion Based Pre Calculation" plugin work, some values of the simulation itself and of the nodes have to be adjusted temporarily. These adjustments involve the validation of the whole simulation time to the pre-simulation time of the "Motion Based Pre Calculation" plugin, because the mobility models plugins maximally calculate till the end of the simulation time. This is not problematic as long the pre-calculation of the starting points are in the range of the whole simulation time period, because now the position of a node at a particular time point can be determined easily. But as soon as the pre-calculation time takes longer than the simulation time, a mobility model will only calculate its paths till the end of the simulation time period and not further. That is why the simulation time of the NS-Mapper has to be temporarily exchanged with the pre-calculation time of the "Motion Based Pre Calculation' plugin.
Additionally, every random node has to perform the mobility model of the pre-calculation temporarily, as long as the calculations last, because the real mobility motion of a mobile node can differ from the mobility motion of the pre-calculation plugin. For this, of course, the pre-calculation plugin has to store the mobility model of a node and has to set it back later.
The temporary changes are not the crucial point by processing the pre-calculation model, because every single node is individually adjusted, calculated and can be again converted into its original state later. The occurring problem is that a faultless course of events of a chosen mobility motion plugin can not be guaranteed in all cases. If an error occurs during the performance of the calculations, an exception handling will prevent its implementation, which will result in a finally block of the pre-calculation plugin. For this, the finally block always will set back all values to the original state, e.g. the original simulation time and the original values of the nodes.
Otherwise the plugin works in the previously described frame of the definitions. The source code of the plugins can be looked up at

```
mapper.plugins.randomAlgorithms.startPointPlugins.
    MotionBasedPreCalculation
```

4.7.2 Random Movement

In this section the "Gauss Markov" plugin of the category mobility motion is described. At first we will explain, which values the plugin needs to be executed regularly, and afterwards we will describe in 4.7.2 how it works and which adjustments were performed to make it work in the NS-Mapper.

Available Options

The "Gauss Markov" mobility model plugin offers the user the following options, see figure 4.6. The first value affects the "linearity" of the function. It presents a percentage, whereas only natural numbers from 0 to 100 are permitted. It declares how linear a node is moving. If the linearity is 100 percent, the movement and the speed of the Gauss Markov algorithm is linear, and not influenced by a random number. If it is zero, the direction and the speed values are completely random based.
The option "Time Interval" defines the time period, in which the position values x, y, the directions and the speed in a loop have to be calculated before being assigned to their definitive position in the simulation. The longer the chosen interval is, the bigger is the distance between the position values x and y.
The option "Average Speed" defines the average speed of a node. The values are calculated by the Gaussian random calculation. The distribution of this function is defined by the two values "Min Speed Distribution" and "Max Speed Distribution", with which the velocity can be demarcated.
The following two options "Direction Distribution" for min and max define the random based directions. Usually values between -1 and 1 are chosen. If higher values are used, the deviation is increased in regard to the chosen factor. In general, these values should not be changed.
The last option contains the "Edge Distance Repulsion". For the NS-Mapper it has to be possible to pass the border of a MovementField. Thus, this value can affect the repulsion from its borders. The higher this value is set, the bigger is the distance to the particular border, in which the repulsion is already effective (repulsion buffer zone). This value has to be adjusted to the topology of the scenario. If a scenario contains many small MovementFields, the repulsion distance should not be set high. Also if it contains large MovementFields, this value should be set higher, otherwise the random based nodes cannot change their MovementField anymore, because the

repulsion influences the movement of the nodes too early. The distance from the node to the border is defined in pixels.

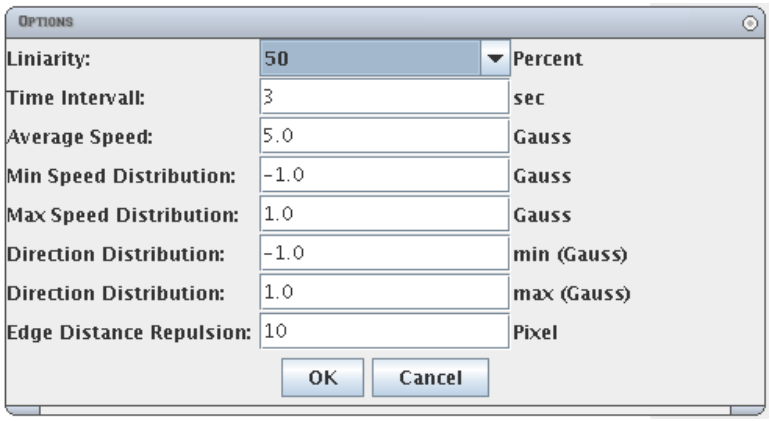

Figure 4.6: Gauss Markov Options Dialog

Implementational Aspects

Some corrections had to be performed on the Gauss Markov algorithm to maintain its usability in the NS-Mapper. On the one hand, the complex structures with its limited movements, generated by the NS-Mapper, are responsible for this need, because the Gauss Markov algorithm originally was only developed for rectangular scenarios, whereas the movements in the NS-Mapper are limited by partially very complex polygons. This causes that the mobile nodes, processed by the Gauss Markov algorithm, are repelled from borders to prevent drastic directional changes. However, a mobile node in the NS-Mapper has to be capable to pass from one MovementField to another, if they touch.
As already detailed explained in chapter 2.3.2, the Gauss Markov algorithm prevents hard changes of the direction of a mobile node, by repelling it from the borders of the scenarios. For the new adjustment of the direction it uses the opposite angle, whereas in a rectangle scenario only 8 opposite directions exist, see figure 4.7, point a. The simple topology of a square requires only a calculation of the distance to the x or y axis. Therefore, calculations of angles become redundant, because only eight repulsion angles emerge. This is due to the fact that the borders of the area can only appear horizontal or vertical. But the NS-Mapper constructs complex polygons,

thus, the opposite angle during a repulsion of the nodes may possess a big variability of directions. For every line, the angle of repulsion has to be calculated separately, see figure 4.7, point b. Of course, this increases the complexity of the Gauss Markov algorithm enormously. In addition, for a complex polygon it has always to be calculated the smallest distance from a mobile node to every border of the MovementField, on which the mobile node moves, as demonstrated in figure 4.7, point c. Therefore, all sections of a polygon have to be searched for all possible distances stringently.

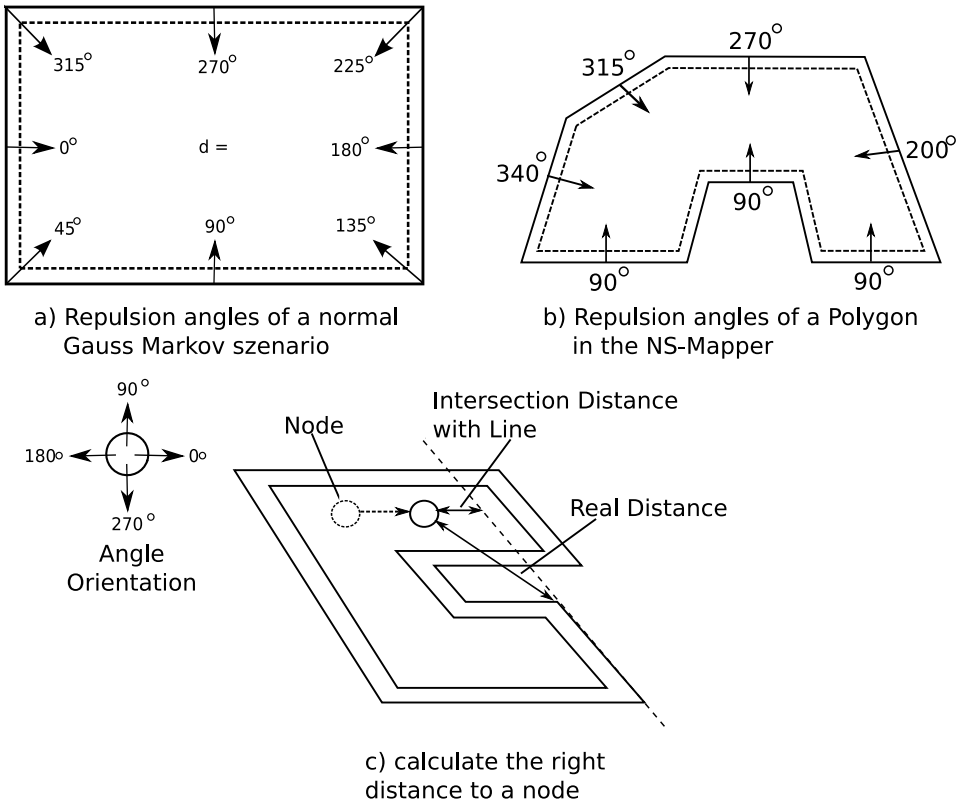

Figure 4.7: Line Calculations

Furthermore, the algorithm has to accept that it has to be possible for the nodes to leave a MovementField to cross to another, without loosing the aspect of repulsion from the border. This is achieved during the implementation of the plugin by balancing the step range and the direction linearity of a single calculation step. The step range is defined as the number of calculation loops of the Gauss Markov algorithm and its speed. The more often the loop is calculated, the more probable is the crossing to a

new MovementField. A suitable step range has to be determined during the tests in chapter five.

4.7.3 Random Traffic

This section deals with the "BitTorrent Topology" plugin of the category random traffic. At first we will describe, which values are necessary to execute the plugin, and afterwards, we explain in 4.7.2 how it works and which adjustments had to be performed.

Available Options

The options, which are available for the BitTorrent plugin, are divided in three parts: Tracker, Client und Connection options. Several wireless nodes belong to the same group, if they share, for example, their Traffic OptionsKeeper object via reference. To generate BitTorrent nodes, which all belong to the same group, the nodes have to be defined simultaneously. This can be accomplished by stating the number of nodes in the "Add Node" dialog. Thereby, all nodes share the same OptionsKeeper automatically. The tracker options have the following configuration capabilities: They are defined by the option "Maximal Trackers", which states the maximal number of the allowed trackers. This number is in general freely scalable, but must not exceed more than half of all defined BitTorrent nodes of one group. If a higher number of trackers is chosen, this number is automatically truncated to the highest allowed number. This is performed because it is not sensible to possess a higher number of trackers than clients, because maximally only as many connections can occur as clients exist.
The option "Tracker Upload" defines the upload, which a tracker permits. The maximal range, that a tracker is able to perform, is declared in KB/s and can be optionally distributed evenly among all connected nodes.
The Client Options have the following configuration capabilities:
The JCheckbox "Bind on own Tracker" ensures that a client can only connect to a tracker, which belongs to its own group of nodes. How to define a group of nodes was explained previously.
The "Average Upload" declares the average upload, which is permitted by the client. The whole upload, which the client can use, is distributed evenly among all connected nodes. But this is only valid, as long the number of

connected nodes does not exceed the number defined by the option “Maximal Download Clients”.
The “Average Active Clients” indicates the average number of nodes, which are chosen to participate actively in the traffic generation. Valid values range between 0 and 50 per cent of the nodes, serving as clients.
The “Leech Time” determines how long a client has to download data from other nodes, until he himself has received enough data to be accepted to participate himself actively in the exchange of data. This means that after this time other clients can connect to it to receive data from it.
The connection options have the following configuration capabilities: “Packet Size” defines the size of the BitTorrent packets. BitTorrent demands that the size of the packet always has a basis of 2 with an exponent <=6. This has to be abided.
Furthermore, the “Time Range Options” defines the “Start Time” and the “Stop Time” in the range of the simulation time, indicating the frame, in which the exchange of data in the BitTorrent network is allowed. As usually, this time is defined in seconds.

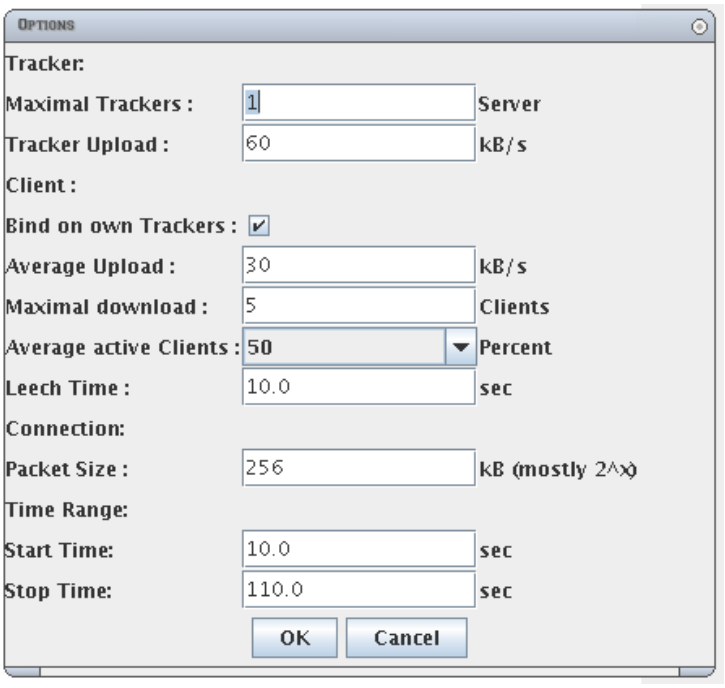

Figure 4.8: BitTorrent Options Dialog

Implementational Aspects

The sequence of the generation of a BitTorrent network is executed as follows:
At first all nodes have to be filtered and collected, which belong to the BitTorrent network. Nodes, which are either appointed to none or to another traffic plugin, of course, do not participate. Subsequently, the number of trackers out of every list is chosen in a random based way, and all remaining nodes form the group of clients. This shall prevent that not always the same nodes function as trackers in the different simulations. Then the start time of those clients is specified, which are allowed to connect to a tracker. If a node is not allowed to connect because of its percentage of "Average Active Clients" of the options, it does not receive a start time and therefore, does not participate as an active client in the BitTorrent network. The random start time is determined by a special calculation, which guarantees that the connections do not accumulate in the middle part of the simulation. The possible start time is defined in the range of the doubled simulation time, whereas half of this time is placed at the beginning of the simulation time and the other half at the end of the simulation time. If a start point is randomly calculated outside of the real range time, it gets its real start time near to the beginning of the simulation, if the random start point is smaller than the simulation time. In the opposite case, the start time is near the end of the simulation time.
After that, the traffic is calculated, which is generated between the trackers and the clients, or between the clients themselves, respectively. In the BitTorrent topology of this plugin, the tracker is always the owner of the data, which the clients want to receive. For this, the client passes three states:

State 0 The tracer is inactive and expects its activation.

State 1 This condition is called Leech Time. The client can download data from the tracker, but does not exchange data with other clients. Its download volume depends on the upload volume, which the tracker permits and the number of clients, with which it has to share the supply.

State 2 The client has received enough data from the tracker. Now he can

also exchange data with other clients. Every client tries to receive as much data as is permitted by the tracker and by other clients. The client distributes the upload volume evenly among all other clients, which have connected to it, up to the maximal allowed number of connections of clients.

As already addressed in chapter 2.3.3, we had to adjust the implementation of the BitTorrent algorithm, because an implementation in a simulator as the NS-2, which generates the traffic traces of a scenario, actually does not need to have an implementation of this protocol. These do not even have to exist necessarily, as for example in the NS-2. The BitTorrent topology is therefore only approximated. This is, e.g., expressed in the fact that at the beginning of the transfer, no exchange of the clients topology takes place, which usually would be the first thing to happen. Additionally, at the beginning of the simulation it cannot be decided, whether a client possesses a connection to its tracker. Of course, this makes the Leech Time and its resulting traffic between the clients redundant. But as we only intend to approach the traffic, and basically the traffic topology is crucial, we have finally decided for this selection. The herein implemented plugin can be adjusted easily to the pre-existing extensions in a simulator as soon as they will exist. As long as these extensions do not exist, the exchange of data in the NS-2 is simulated as constant bitrate on TCP basis, and is translated into a trace file by the export plugin of the NS-Mapper.

4.8 Summary

In this chapter we have shown on the basis of many code examples, how the data structure, parts of the framework and the random based plugins were implemented in the NS-Mapper, as well as how a developer can use them for his own random plugins. In detail, the single interfaces of the data structure including their hierarchic construction were discussed. It was explained precisely how to reach relevant data of this structure (section 4.2). Furthermore, the most important interfaces of the framework were explained. It was described how they were implemented and, of course, how to use them (section 4.3). The most focused aspects were: how the single object structures are cleaned up, how to draw in the PaintingArea, how mouse and keyboard events are caught, how to access marked elements, and finally

the most important: how to link plugins, how to assign them options and how to adjust their movements to the limitations of the NS-Mapper. We have clarified how the plugins are integrated in the programme structure of the NS-Mapper. At the end, we have described the implementation of the random based plugins and what special features were necessary for their implementation, to guarantee a smooth course of events, still following their exact algorithmic definitions.

Chapter 5

Evaluation

This chapter deals with the planning, the execution and the analysis of test scenarios, which allow a direct comparison between the newly developed random plugins and the already implemented plugins of the former study thesis. The random plugins, which will be evaluated regarding their effectiveness, will be in detail: at first, the "Motion Based Pre Calculation" plugin for the start point, which will be compared with the "Bounding Box Random" plugin, secondly the "Gauss Markov" algorithm, representing the category of mobility models, which will be compared with the "Random Waypoint" algorithm, and finally the "BitTorrent Traffic" topology, which will be compared with the "Rudimentary Traffic" plugin.
We will introduce the general, basic structure of the scenarios, which will be maintained in all test runs to guarantee a comparable analysis. Afterwards, every individual scenario will be presented in detail, to explain which algorithm is used for it. At the end follows the presentation of the data and their analysis.

5.1 Simulation

For the simulation a setting is used (as already utilized in the study thesis) that allows to draw conclusions about the quality of specific scenarios, and thereby their used random plugin, by analyzing a limited, though representative, number of single scenarios. Additionally, by means of statistic mean values of the analyzed data, particularly significant scenarios can be recognized and filtered, which either produce exceptionally good or exceptionally bad results. This way it is not only possible to achieve a representative in-

terpretation of the analysis, it is also possible, for example, to evaluate how a worst case scenario looks like and which characteristics it shows.
While generating random based scenarios with the NS-Mapper, it is without any problems possible, to keep special basic values of a scenario constant, for example the maintenance of start positions, whereas the random based movements or the traffic may vary. An advantage of this procedure is to be able to draw more precise conclusions in regard to a specific random plugin. Additionally, this time we can use a procedure for the construction of scenarios, which last time was not possible because of the absence of the ability to mix different random plugins. The major reason why the NS-Mapper was developed was due to the fact, that different random procedures could only be connected by means of a massive intervention of the user, because the former used software did not support this. Now the NS-Mapper has actually the ability to group various random procedures and to mix them in one single scenario. We will use this feature during the construction of scenarios.
The probably most complicated, but still most important aspect during the planning and the analysis of scenarios, which will be directly compared with each other, is that the scenarios have to achieve an acceptable level of direct comparability. This should not be a problem for comparing a random start point model with a random mobility model. But the generation of random based traffic is based on a completely different complexity of its topology, and also these have to be nearly comparable for analyzing them. The question that has to be answered is how to construct a comparable traffic scenario, which remains faithful to its natural characteristics without portraying or imitating another topology, only to achieve a level of comparability. Because then, one would compare the same traffic topologies with each other, which are only constructed by different methods, and this kind of analysis would be worthless.

5.1.1 Basis Environment

For every category of random algorithms, start point, movement and traffic, 20 scenarios are constructed. Ten of them will be assigned to the plugins, formerly developed during the study thesis, the other ten to the plugins, developed in this diploma thesis.
The used base environment, with which the random plugins will be tested,

has the size of 500 times 500 meter. Three nodes, which are connected to each other, are analyzed directly. The distance from the node in the bottom left corner to the node in the upper middle has a length of 380 meters, and the distance from the bottom left node to the upper right corner is 621 meter. These nodes do not move during the whole course of simulation. Their distribution is presented in figure 5.1. Because these three nodes do not possess a direct connection to each other, their packets have to be forwarded by other mobile nodes. The simulation time for each scenario takes two minutes, in which the nodes establish a connection within ten to 110 seconds.
To achieve a large enough number of additional mobile nodes in the network, which can forward packets, 20 further ad-hoc nodes are added, with an average speed of a pedestrian between 3 and 5 kmh, which are able to move in the base environment. The scenarios, in which the random start point and the movement plugin of the NS-Mapper are evaluated, possess - apart from the tree basic nodes - no further traffic or background traffic, respectively, because this might influence the results so much that it would be not possible to decide if the changes are caused by the background traffic or by the plugin itself. To be able to exclude any disruptive factor, the simulations are kept as simple as possible. Only in the scenarios for testing the traffic models, additional mobile nodes will also partially generate their own background traffic, which will be integrated in the evaluation, but will not be analyzed directly.

The node configuration, which is set as default values in the NS-Mapper, serves as a basis for all scenarios. But whereas the first two categories, the "Start Point" and the "Mobility Motion" scenarios, use a data rate of 2Mbs, to make us recognize small fluctuation, the "Traffic Topology" is tested by scenarios with both, a data rate of 2Mbs and 54Mb/s, respectively. 2Mbs is used to be able to determine exactly if the two compared algorithms are almost equal. 54Mb/s is used to increase the traffic amount in these scenarios. This way, we can judge how the BitTorrent Topology stresses the network. For further information regarding the settings, which can be used in the NS-2, refer to the documentation [7] or the former study thesis [8]. For a brief explanatory enumeration of the functions, which are used in the simulation, we recapitulate:

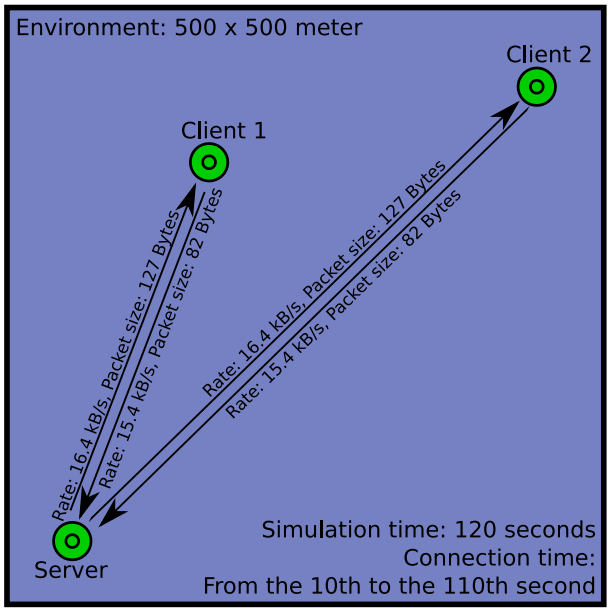

Figure 5.1: Basis Simulation Environment

adhocRouting

The "adhocRouting" offers four, in the NS-2 officially supported, routing protocols: AODV, DSR, DSDV and TORA. In this context only the AODV Protocol (Ad hoc on Demand Distance Vector Protocol) will be used.

llType

The Link-Layer object is responsible for the simulation of the Data-Link-protocol. It contains the fragmentation of the packets and the reassembly by a reliable link-protocol.
Another important function of the Link Layer in the NS-2 is to write the MAC-address of the recipient in the header of the transmitted packet. In the currently implemented form in the NS-2, it handles two different tasks: 1. to find the IP-address of the next hops and 2. its transfer to a correct MAC-address (ARP). In the simulation the "LL" type will be used.

macType

The NS-2 is familiar with several MAC-tyes, including Mac/802_11, Mac/C-sma /Ca. Roughly speaking, it handles the settings of date rate, preambel-

and header length, the SlotTime in the medium, etc. The IEEE 802.11a Wireless Standard of the Institute of Electrical and Electronics Engineers (IEEE) is used.

ifqType

The "ifqType" defines the used queue of the simulation. It contains: Queue/DropTail, Queue/DropTail/PriQueue. Up to now, only the Queue/DropTail/PriQueue was successfully tested in the NS-Mapper.

ifqLen

The "ifqLen" defines the length of the queues. The default setting of the length of the queue is 50 packets.

antType

As 'AntennaType" only the Antenna/OmniAntenna can be used, therefore it is chosen necessarily for the simulation.

propType

Possible propagation models, which can be used in the NS-Mapper are the "Propagation/TwoRayGround", which will be applied for the simulation, as well as the "Propagation/Shadowing" and the "Propagation/FreeSpace".

phyType

For the physical type in the NS-Mapper the following functions are at disposal: Phy/WirelessPhy, Phy/Sat. Because the scenario consists of an ad-hoc network, it is simulated by the "Phy/WirelessPhy".

channel

The "WirelessChannel" is usually the applied channel. However, the NS-documentation is not provided with further information regarding its way of function or usage.

The following table 5.1.1provides an overview of the settings, which are used by the Ad-Hoc nodes of our scenarios.

Function	Subfunktion	Value
addressingType		flat
llType		LL
macType		Mac/802_11
ifqType		Queue/DropTail/PriQueue
		CMUPriQueue
ifqLen		50
phyType		Phy/WirelessPhy
adhocRouting		AODV
propType		Propagation/TwoRayGround
antType		Antenna/OmniAntenna
channel		Channel/WirelessChannel
topoInstance		load_flatgrid 500 500
agentTrace		ON
routerTrace		ON
movementTrace		ON

Listing 5.1.1: Node configuration in the Scenarios

5.1.2 Basis Environment Traffic Model

The basic traffic, which we generate in the scenarios, is based on three mobile nodes, which play a computer game against each other. One node will take the position of the server, two other nodes those of the clients. In the scenarios also the server represents a player, which, however, transmits different data packets.
This used traffic model is based on the “Network Gaming Traffic Modelling” (Literaturliste [9]) and produces an average transmission rate from the server to the client of 16,4 k/bit per second, and from the clients to the server 15,4 k/bit per second. The size of the packets of the server is averagely 127 bytes, that of the client 82 bytes (see image 5.1).
The scenarios adopt this model in a slightly modified way, by means of an UDP-connection, which transmits the packets, applying a constant bit rate. That means the model ignores variations of the size of the packets, which usually appear in a game of this kind. But a corresponding setting does not exist in the NS-Mapper for achieving the realization of variable packet sizes.

In other games, the distribution of the size of the packets would be different: Assuming an average packet size, transmitted by the server, of 127 bytes, 99% of the packets would have a size less than 259 bytes, the rest would not exceed 1500 bytes per packet, whereas for the clients 99% of the packet sizes would be between 60-110 bytes. These variations are straightened by the constant bit rate, which is used in our simulations.
To guarantee a good course of game, it is necessary that most packets reach the client or the server, respectively, within 40 ms. These values are the highest allowed latency period; they correspond to a game via analog modem in the "Network Gaming Traffic Modeling". For a human being a RTT is only noticeable after 100-150 ms, but player with a lower RTT than 100 ms achieve an essentially higher efficiency [9].
Table summarizes 5.1.2 the values of the traffic model once again:

	Server	Client
Source		
Agent	UDP	UDP
Sink		
Agent	LossMonitor	LossMonitor
Application		
Application	Traffic/cbr	Traffic/cbr
Packet Size	127 Bytes	82 Bytes
Rate	16.4	15.4

Listing 5.1.2: The bidirectional traffic between the three core nodes

5.2 Random Start Point Environment

This section presents the results of the comparison of the start point algorithms. As explained, in addition to the tree nodes, which were introduced in section 5.1.1, 20 other mobile nodes were added in the scenario. These 20 nodes were once distributed by the "Bounding Box Random" algorithm and another time by the "Motion Based Pre Calculation" algorithm. The saved NS-Mapper files, the trace files and their analysis can be found on the CD under "scenarios/SPP", where the directory "SP-BBR_MM-RW" contains the data of the "Bounding Box" and the directory "SP-MBPC_MM-RW"

offers the data of the “Motion Based Pre Calculation”.

5.2.1 Configuration of the compared Algorithms

This scenario was set up to compare the start point algorithms “Bounding Box Random” with the “Motion Based Pre Calculation random”. The second algorithm was already explained thoroughly in section 2.3.1, but it should be briefly mentioned that also this algorithm uses the ”Bounding Box Random” for performing the first node placement. Then the “Random Way Point” mobility model is used with a pre-calculation time of 120 seconds to distribute the nodes newly in the scenario. The default settings are used for the configuration of the mobility motion algorithm.
The ”Bounding Box Random“ algorithm is based on the common distribution of nodes, which is usually used for the construction of scenarios. For the common distribution, a random value is calculated for the x and y position, which is located inside the borders of the scenario. Because the scenarios of the NS-Mapper can place their nodes only within the MovementFields and therefore, possess only a restricted right to place mobile nodes, this common distribution is corrected: This happens by firstly calculating the exact surface of every MovementField. Depending on the calculated size, the probability to place a node on it is determined. The bigger the MovementField is, the higher is the probability of being chosen. When a MovementField is selected, it is tried to place the node in the area of its bounding box. The bounding box has to be used, because the MovementField might be a convex polygon and therefore, its placement is considerably more complicated than for a common rectangle. Figure 5.2 demonstrates which cases might appear. The placement of a mobile node within the bounding box will be repeated as often as a regular placement has been found, which refers to a place inside the MovementField.

5.2.2 Analysis

The analysis of the data did not reveal any significantly important differences in regard to the node placement, between the “Bounding Box“ (**BBR**) and the “Motion Based Pre Calculation“ (**MBPC**). In the following table the single connections between server and client 1 or client 2, as well as the connections between the clients back to the server are demonstrated in the horizontal lines. Vertically the categories of the single scenarios with their

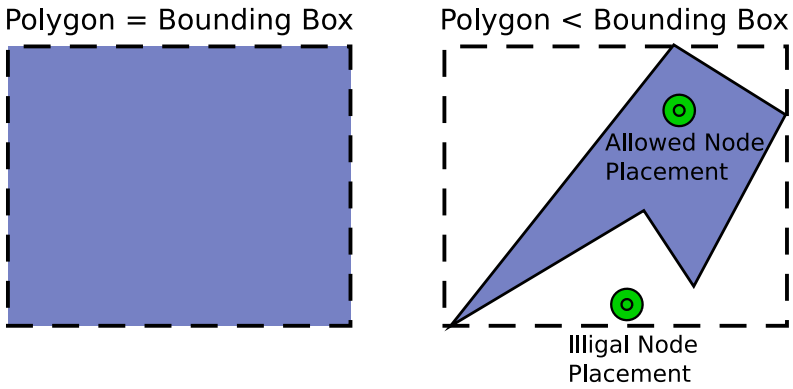

Figure 5.2: Bounding Box Random

sent and lost packets (**pk**) are shown, as well as their delay is depicted.

	Ser -> Cl_1	Ser -> Cl_2	Cl_1 ->Ser	Cl_2 -> Ser
BBR				
Send	1607 pk	1606.2 pk	2346.1 pk	2332.1 pk
Lost	1.7 pk	9.2 pk	6.7 pk	16.4 pk
Lost/Send	0.10%	0.57%	0.28%	0.70%
Delay	0.014 sec	0.0208 sec	0.015 sec	0.030 sec
MBPC (with scenario 7)				
Send	1585 pk	1580.9 pk	2346.6 pk	2288.4 pk
Lost	1.6 pk	8.5 pk	39.1 pk	45.8 pk
Lost/Send	0.10%	0.53%	1.66%	2%
Delay	0.017 sec	0.023 sec	0.019 sec	0.047 sec
MBPC (without 7)				
Send	1610.5 pk	1609.5 pk	2346.1 pk	2282.4 pk
Lost	1.7 pk	7.3 pk	3.8 pk	11.3 pk
Lost/Send	0.11%	0.45%	0.16%	0.49%
Delay	0.015 sec	.019 sec	0.014 sec	0.035 sec

Listing 5.2.2: Comparison of average Send and Lost packets, with its delay in the Random Start Point Environment

In the scenarios, no packets of the three analyzed nodes were dropped and only very few got lost during the transmission. It should be noticed that

within the analyzed trace files of the “Motion Based Pre Calculation” algorithm one scenario (number 7) exists, which shows a particularly higher number of packet losses. As a result, the average of the whole analyzed scenarios of this category is significantly worsened. However, if this trace file is excluded from the analysis, the average data of the lost and dropped packets in comparison to the “Bounding Box” random algorithm are approximately equal. That is why the analysis of the “Motion Based Pre Calculation Algorithm” is quoted twice in the table, once with and once without the trace file number 7.
In the evaluation of the send and lost packets sometimes the one and sometimes the other algorithm reached better results, so that no clear statement can be made which algorithm is better. Also the quantitative deviations between the scenarios of this category were not high enough to make conclusions of a difference in behaviour. The whole values for the “Motion Based Pre Calculation” algorithm, without the mentioned particular tracefile, were only slightly better than the “Bounding Box Random” algorithm. All data about send, dropped, lost packets, as well as all further data are included on the cd, saved in the directory for the start point plugin, for details refer to Appendix B.
Also the direct comparison of the throughputs between scenario 10 of the ”Motion Based Pre Calculation“ and scenario 8 of the ”Motion Based Pre Calculation“, which present roughly the mean value of all constructed scenarios in this category, revealed that the values are approximately congruent, see figure 5.3 and 5.4. In this picture, horizontally the simulation time is declared by the unit of 120 seconds, vertically the unit is 10^4 bitssec [1]

The assumption that the centring of the nodes[28] in the middle of the scenario, produced by the ”Motion Based Pre Calculation“, provokes another behaviour during the transmission, could not be confirmed by the constructed scenarios. Even if the average values of the Pre Calculation algorithm, excluding scenario 7, were slightly better than those of other placement methods, the results of the dropped and lost packets, the delay and the throughput were not significant enough. Surely, it could be considered that in a scenario, in which also other nodes generate background traffic, the differences might become more important; nevertheless, our sim-

[1]The analyzed data could not be presented in the same picture, because the tool for analysis (TraceGraph) did not support a merge of graphs.

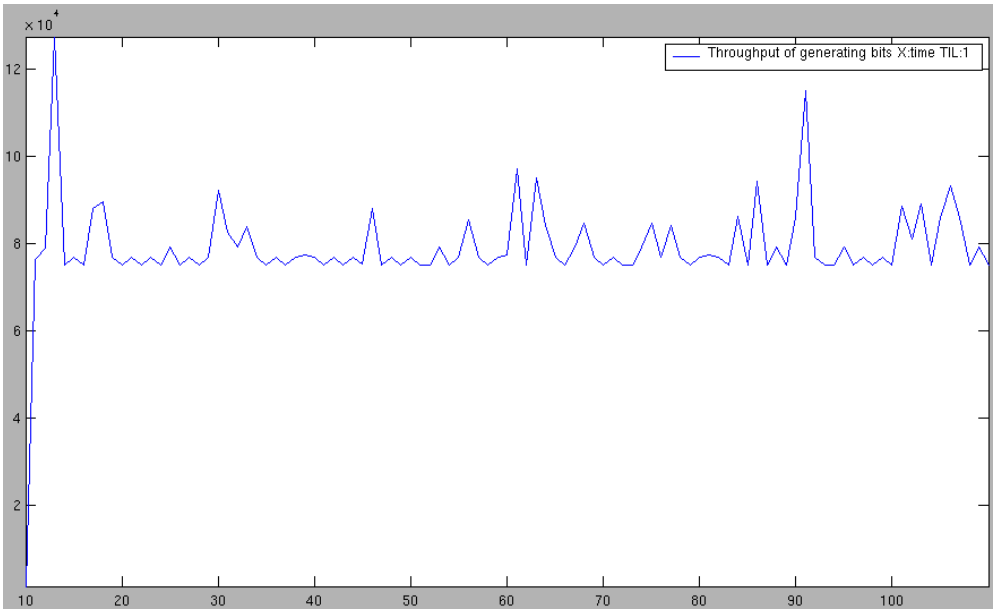

Figure 5.3: Bounding Box Random Throughput

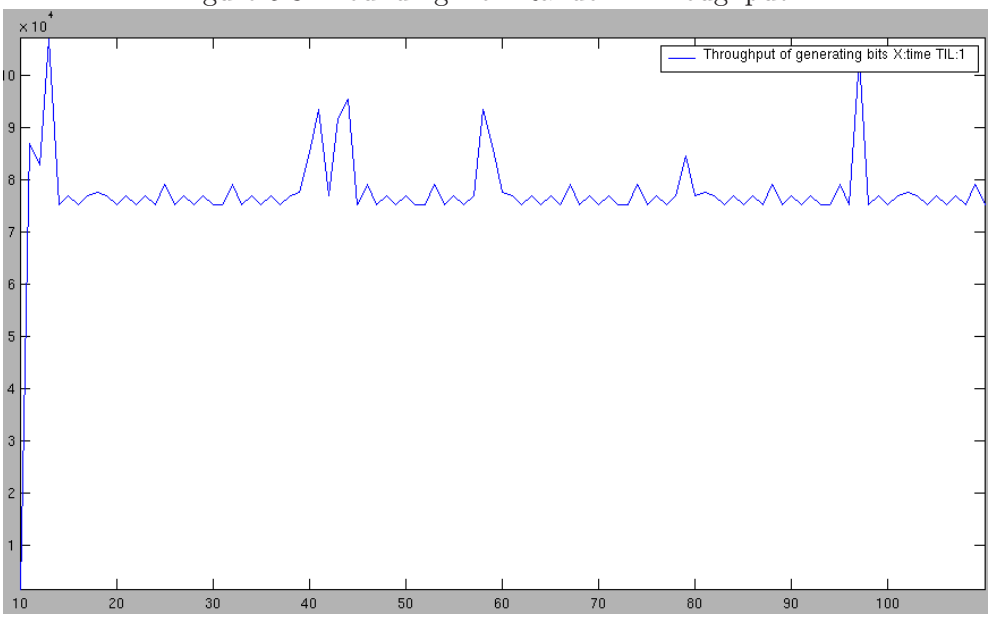

Figure 5.4: Pre Calculation Throughput

ple method of analysis could not confirm this. The percentage deviation between the generated and the lost packets were, for instance, not large enough and the average is 1/10 per cent.
Also the assumption that the Pre Calculation algorithm distributes the forwarding of packets more uniformly to all mobile nodes could not be confirmed after the examination of the data by TraceGraph.

5.3 Random Mobility Model Environment

This section presents the results of the comparison of the mobility motion algorithms. Again, in addition to the tree nodes, that were introduced in section 5.1.1, 20 other mobile nodes were added in the scenario. These 20 nodes were once moved by the "Random Way Point" and another time by the "Gauss Markow" algorithm, respectively. The saved NS-Mapper files, the trace files and their analysis can be found on the CD under"scenarios/MMP", where the directory "SP-MBPC_MM-RW" contains the data of the "Random Waypoint" and the directory "SP-MBPC_MM-GW" offers the data of the "Gauss Markov Algorithm".

5.3.1 Configuration of the compared Algorithms

For the first used algorithm, the "Random Way Point", the default settings of the plugin were adopted for the scenario. The velocity of the nodes was between 3 and 10 kilometres per hour, which corresponds approximately to the speed of a pedestrian. The distribution between the minimum and the maximum value of the velocity is performed by the common distribution. The minimum and the maximum waiting time of the nodes during the single movement sections is between 1 and 5 seconds, whereas these values are subject to a Gaussian random distribution, and 2,5 seconds form the average value.
The configuration of the "Gauss Markow" algorithm is moved with a direction linearity of 75%, see 2.3.2. The loop interval for the calculation of velocity and direction is 3 seconds (every interval counts for 1 second). The average velocity of the nodes is defined as 5 km/h, and the deviation of the velocities is calculated by a Gaussian distribution, where the deviation is between -3 and 3. The repulsion from the border is defined as 10 pixels for

the scenario.

5.3.2 Analysis

In the calculated trace files the "Gauss Markov" algorithm has averagely always obtained better results than the "Random Waypoint". As already noticed in the former comparison of the start point algorithm, also here no packets of the examined nodes were dropped and only very few got lost. Although the number of the lost packets is very low with both algorithms, still the improvement of the "Gauss Markov" algorithm reaches approximately 40% for all examined node connections (see table GM vs. RW, percentage improvement), which definitely is significant. Additionally, the delay times of the "Gauss Markov" algorithm turned out to be lower, even if here the results are closer to each other.

	Ser -> Cl_1	Ser -> Cl_2	Cl_1 ->Ser	Cl_2 -> Ser
Random Waypoint				
Send	1611.5 pk	1613.1 pk	2345.8 pk	2323.3 pk
Lost	2.4 pk	6 pk	6.9 pk	18.1 pk
Lost/Send	0.14%	0.99%	0.29%	0.77%
Delay	0.013 sec	0.023 sec	0.018 sec	0.04 sec
Gauss Markov				
Send	1612.4 pk	1613 pk	2345.1 pk	2334.5 pk
Lost	1.3 pk	9.8 pk	4.0 pk	8.6 pk
Lost/Send	0.08%	0.60%	0.17%	0.36%
Delay	0.012 sec	0.021 sec	0.013 sec	0.04 sec
Percentage Improvement of the Lost Packets between GM and RW	-42.85%	+39.39%	-41.13%	-53.24%

Listing 5.3.2: Comparison of average Send and Lost packets, with its delay in the Random Mobility Model Environment

The better result of the "Gauss Markov" was also confirmed by the direct comparison of all generated packets in all scenarios, including those, which derived from the routing of the AODV protocol. In the scenarios, which were generated by the "Random Waypoint" algorithm, 2,07% of the packets were dropped or got lost. Scenarios, generated by the "Gauss Markov"

algorithm, only showed a drop or loss of 1,33%, which is an improvement of 35,74% in regard to the net stability.

	Random Waypoint	Gauss Markov
Send	8850.8 pk	8468.5 pk
Dropped	109.9 pk	66.9 pk
Lost	73.9 pk	45.9 pk
Dropped and Lost	183.8 pk	112.8 pk
Percentage (Droppend and Lost)/Send	2.07%	1.33%

Listing 5.3.2: Comparison of average send, dropped and lost packets from all packets made in the whole environment of the Random Mobility Model

However, the comparison of the throughput of the analyzed nodes did not reveal any big differences, as already obtained in the previous category, because also here the additional 20 nodes did not generate additional traffic. Who is interested in the precise data, may refer to the *.mat Files of TraceGraph on the CD.
As assumed, the “Gauss Markov” algorithm scored better on average than the “Random WayPoint”. This might be due to the fact that the nodes of the second algorithm move much straighter and therefore, are sooner not considered optimal anymore in the routing of the AODV protocol. The probability that the mobile nodes remain longer in a particular area is clearly visible when evaluating the data. Therefore, one should consider which courses of movement a particular scenario resembles, when planning his own scenario. Furthermore, it is necessary to point out that the “Gauss Markov” algorithm is capable of imitating the “Random WayPoint” or at least that it can achieve comparable results, by influencing its direction linearity and other settings. Therefore, it provides a clearly more variable behaviour.

5.4 Random Traffic Environment

This section presents the comparison of the traffic topologies. In addition to the tree nodes, introduced in section 5.1.1, this time 5 additional mobile nodes were added in the scenario, which either generate “Rudimentary Topology” traffic or “BitTorrent” traffic. 15 further mobile nodes moved

additionally in the network. The saved NS-Mapper files, the trace files and their analysis can be found on the CD under “scenarios/TP”, where the directory “/SP-MBPC_MM-RW_TP-RT” contains the data of the “Random Traffic” and the directory “SP-MBPC_MM-RW_TP-BT” offers the data and scenarios of the “BitTorrent”.

5.4.1 Replic of the BitTorrent Traffic

To generate a comparable traffic by the “BitTorrent Topology” and the “Rudimentary Topology”, we have to recall how these two topologies are constructed. This knowledge allows us to prepare calculations, which are necessary to obtain a comparable number of mobile nodes, as well as a comparable traffic, which generates these nodes. These calculations have to be as precise as possible, because we have want to generate the “Rudimentary Topology” traffic to run completely automatically.
We have the BitTorrent topology with n nodes, whereas one of these nodes represents the tracker, see figure 5.5. All other nodes, the clients, possess a connection to each other, see also 4.7.3 with the “Implementational Aspects” of the BitTorrent Topology. In this topology the users usually share very large files, but the calculation times are rather short. Thus, in these scenarios the BitTorrent clients connect to a tracker at a random time and maintain this connection till the end of the entire simulation time.
On the other side, we have the inhomogeneous, rudimentary traffic of the “Rudimentary Topology”, which possesses a number of m nodes. All of these nodes show the same behaviour. We cannot identify or separate any of these nodes, which act independently of the BitTorrent nodes. These mobile nodes connect arbitrarily to each other. The rudimentary topology, in contrast to the BitTorrent, establishes its connections at random time points during the entire simulation time, and also closes them at a random time points. Because of that different behaviour between the two topologies, we need to calculate exact average values of the number of nodes in the network, which generate traffic; and we also need a precise average value of this traffic.

To reproduce the BitTorrent topology with the “Rudimentary Topology”, we have to calculate how many connections are possible to occur on average in the BitTorrent topology. That means, there are altogether n nodes, which have to be connected. The number of connections between

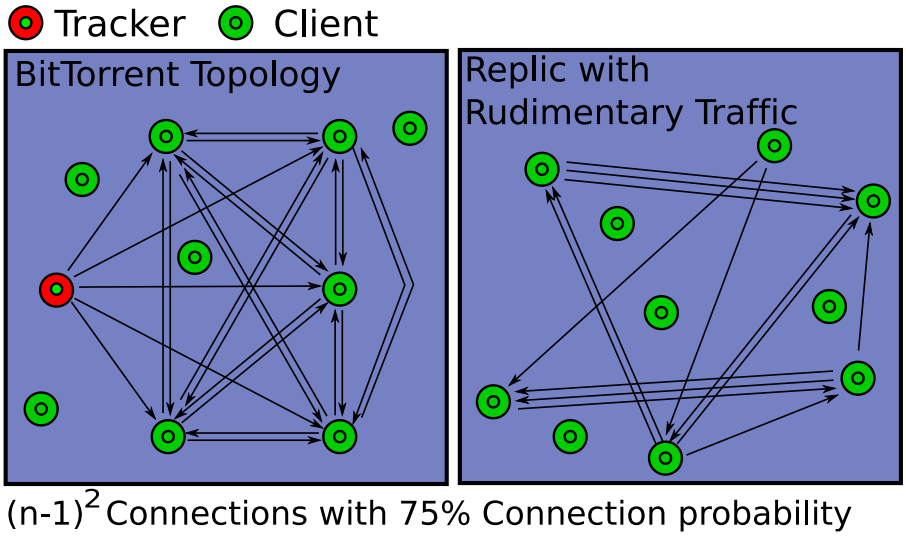

Figure 5.5: Replication of the BitTorrent Traffic

the tracker to the clients is $n-1$, because the tracker is always connected to the clients. Additionally, the clients possess connections to each other. Maximally, all clients are connected to each other. This equals a number of $(n-1)*(n-2)$ connections altogether. This results in a total amount of maximal connections (**MC**) multiplied with the formula of the "Binominal Distribution" of the probability calculus and resuls in

$$n \geq 1, MC = (n-1) + (n-1)*(n-2) = n^2 - 2n + 1 = (n-1)^2$$

connections, which may appear in a network, containing n BitTorrent nodes. Because there is a connection probability of $t\%$ that a client connects to a tracker, we obtain for the "Rudimentary Topology" a probability of

$$n \geq 1, m = (n-1)$$

$$AC = \sum_{r=0}^{m} (\frac{m!}{r! * (m-r)!}) * (\frac{t}{100})^r * (1 - \frac{t}{100})^{m-r} * r^2$$

connections, which is also multiplied with the "Binominal Distribution". Yet, we have to determine the BitTorrent traffic, which averagely charges the network. There is the traffic, which can originate from the tracker, containing an upload volume of x kB/s, as well as $(n-1)$ clients with an upload volume of $ct = (n-1)*y$ kB/s, if we have more than 1 client. Thus, we obtain a maximal traffic of:

$$n = 1; MT = 0$$

$$n = 2; MT = x$$

$$n > 2; MT = (x + (n-1) * y)$$

and average Traffic (**AT**) of:

$$n > 2, m = (n-1)$$

$$AT = (\frac{m!}{1! * (m-1)!}) * (\frac{t}{100})^1 * (1 - \frac{t}{100})^{m-1} * x +$$

$$\sum_{r=2}^{m} (\frac{m!}{r! * (m-r)!}) * (\frac{t}{100})^r * (1 - \frac{t}{100})^{m-r} * (x + r * y)$$

5.4.2 Configuration of the compared Algorithms

Depending on the prior calculations, we now can configure our network. Basically, we insert 15 nodes, which run via the "Gauss Markov" algorithm, so there are completely 20 mobile nodes with the 5 nodes generating background traffic. These mobile nodes do not generate traffic on their own und are only intended for the routing. Additionally, the mobile nodes of the "BitTorrent topology" and the "Rudimentary Traffic" are generated by the "Random WayPoint", so that they can pass through the environment faster.

BitTorrent

We define 5 mobile nodes for the topology of the BitTorrent, whereas one node functions as the tracker. A maximum download volume of 60 kB is assigned to this tracker, which the clients can distribute among themselves. For the additional clients we allow a download volume of maximally 30 kB. To make sure that all clients are connected to each other, we declare the maximum number of connection among the clients to be a value of 5. This is clearly above the number of nodes, which can maximally appear. Because, nevertheless, the nodes amount is relatively small and the probability, that there will be no client at all, is quite big, we set the connection probability of a client up to 75%. The leech time, in which a client is connected only to the tracker, is defined as 2 seconds. The packet size is put to the value 2^7kB = 128 kB. We allow the establishment of connections after the 10th second.

Rudimentary Topology

For the "Rudimentary Topology" we assign 5 mobile nodes, which generate traffic. Because we also declared 5 nodes in the BitTorrent network, which connect to each other with a probability of 75%, in this network we obtain a maximum number of connections of

$$n = 5; (n-1)^2 = 16$$

and an average number of connections **AC** of

$$AC = \sum_{r=0}^{4} (\frac{4!}{r! * (4-r)!}) * (\frac{3}{4})^r * (1 - \frac{3}{4})^{4-r} * r^2 \approx 10$$

Because we can not separate the trackers connections in the "Rudimentary Topology", we have to approximate the probability of connections to the maximum value (= 16) and the average value (≈ 10) of connections of the BitTorrent topology. This is possible if we allow maximally 3 connections for each node with 5 nodes or 4 connections with 4 nodes. The best approximation of the results for maximal connections is

1. $n = 4; MC = n * 4 = 16$
2. $n = 5; MC = n * 3 = 15$

connections, but with 5 connections the traffic is distributed better, thus, we take $n = 5$. Of course, for the 15 connections we have to adjust the average connection probability (**ACP**)

$$ACP = \frac{10}{15} * 100 \approx 67\%$$

. As a result, we obtain the values of the "Rudimentary Topology" by the values "Source Connections = 5", "Connection Probability = 67%" and "Sink Connections = 3".
The number of traffic, which occurs with BitTorrent, is 60kB/s for the tracker, and 30kB/s for the clients each, resulting in an average traffic (**AT**) of approximately

$$m = 4, AT = (\frac{4!}{1! * (4-1)!}) * (\frac{t}{100})^1 * (1 - \frac{t}{100})^{4-1} * 60 +$$

$$\sum_{r=0}^{4} (\frac{4!}{r! * (4-r)!}) * (\frac{3}{4})^r * (1 - \frac{3}{4})^{4-r} * (60 + r * 30)$$

$$AT \approx 148kB/s$$

This has to be distributed among the 10 average connections.
The packet size in the "Rudimentary Topology" is also put to 128kB, to equal the BitTorrent packet size, but with a data rate of $\frac{148}{10} \approx 15kB/s$, where 10 is the average number of connections.

5.4.3 Analysis

As described in section 5.1.1 in this category we performed two kinds of analysis settings, which only differed in the data rate of the scenarios. At first we will discuss the analysis of the scenarios using a data rate of 2Mbs. This setting allows us to draw conclusions of how the traffic in the scenarios differs from each other in the two different topologies and how it reacts in general. Afterwards, we evaluate the traffic using a data rate of 54Mbs. This is the data rate presently used for Ad-Hoc network. Hereby, we can draw conclusions of how strong the peer-to-peer network is charged by the topologies.

Topology Datarate with 2Mbs

When analysing the data, the first thing that was striking positively was that the data of the "Rudimentary Topology" and the "BitTorrent Topology" with 2MBs were very similar to each other, in spite of the necessary approximation of the "Rudimentary Topology" to the "BitTorrent Topology". This proved that the calculations were performed correctly, obtaining a balanced distribution.
The connections between the two clients to the server are almost identical in both topologies (see table 5.4.3 below) and also the connection of the server to client 2 did not differ very strongly from each other. But in both categories, only half the number of packets of the average number of packet is sent, which are ca. 1600 from the server to the clients and 2300 from the clients to the server. This reveals that the network is already completely occupied. Additionally, it is very striking that the connection of the server to client 1 deviates stronger from the other established connections, while the connection in the opposite direction, from client 1 to the server, did not reveal any difference. With the "Rudimentary Traffic" the number of sent packets was 669.2, which is as much as 20.58% less than the number of sent

packets with “BitTorrent”, exposing a difference of 164 packets in all. This cannot be explained by a large number of packet losses, because the number of lost packets only shows a difference of 47.8. It is very surprising that the packets from the server to the client 1 differ so strongly, although both scenarios possessed the same configuration. Perhaps this results are caused by the small number of scenarios, which this time, was only 10 per category. When studying the delay times of the scenario with 2Mbs data rate, as pictured in the table 5.4.3, it is not surprising that under these circumstances, a computer game is not practicable for both scenario forms, neither with the “Rudimentary Topology“ nor with the ”BitTorrent Topology“. As stated when explaining the definition of the base nodes (see section 5.1), it is necessary to achieve a RTT of at least 100-150 msec, whereas the computer playing wireless nodes in these scenarios demonstrate a delay of as much as 1-2 seconds, which is higher than the whole RTT. Even a very small BitTorrent network, in which maximally only 5 nodes acts, already worsens the delay in a dramatic form.

	Ser -> Cl_1	Ser -> Cl_2	Cl_1 ->Ser	Cl_2 -> Ser
Rudimentary Traffic				
Send	669.2 pk	569.0 pk	2121.7 pk	808.6 pk
Lost	34.7 pk	79.0 pk	201.6 pk	137.7 pk
LostSend	5.18 %	13.88%	9.50%	17.02%
Delay	1.183 sec	2.210 sec	1.977 sec	1.619 sec
BitTorrent Traffic				
Send	833.8 pk	650.1 pk	1949.3 pk	783.2 pk
Lost	82.5 pk	103.9 pk	200.2 pk	144.4 pk
LostSend	9.89%	15.98%	10.27%	18.43%
Delay	1.802 sec	2.544 sec	2.544 sec	2.895 sec
Differences (Percent)				
Send	+20.58%	+12.56%	-8.12%	-3.14%
Lost	+57.93%	+23.96%	0.00%	+4.63%

Table 5.4.3: Comparison of the average values of the playing nodes with 2Mbs data rate

Also the examination of the all generated packets in the scenarios for the rudimentary and the BitTorrent topology, confirmed the conclusion that

the calculated approximation was very successful (see table below). In the whole network a similar number of packets was sent. Also the number of dropped packets, meaning those which were rejected by the queue of a mobile node, is very similar to each other and this is additionally expressed by the percentage loss rate.

	Rudimentary Traffic	BitTorrent Traffic
Send	19307.0 pk	19357.8 pk
Dropped	4551.5 pk	4952.9 pk
Lost	86.0 pk	209.7 pk
Dropped and Lost	4637.5 pk	5162.6 pk
DL/Send	24.01%	26.66%

In both categories, the scenarios 5 come closest to the average of all constructed scenarios in regard to their topology. That is why we refer to these when making comparisons to each other. Figure 5.6 and 5.7 demonstrate the throughput of the "Rudimentary Topology". The first diagram indicates the throughput of the server (node 0) to the clients (nodes 1 and 2) and the second diagram shows the throughput of the two clients to the server. The same is demonstrated for the "BitTorrent Topology" by the figures 5.8 and 5.9.
In all constructed scenarios, the throughput varies very strongly. In both topologies, the connections between the server and the clients hardly achieve any periods with a regular, uniform throughput. This is caused by the background traffic. However, client 1 manages to generate a quite uniformly distributed connection to the server temporarily, especially in the "Rudimentary Topology". Indeed, client 1 is located closer to the server than client 2; however, it remains unclear, why particularly this connection shows such a different behaviour, because also this node does not possess a direct connection to the server und its packets also have to be forwarded by other nodes.

It is obvious that the presence of file sharing services in a net, as for instance the BitTorrent, especially in Ad-Hoc networks, causes an enormously negative influence. Even if only few nodes (for example 5, as in this topology) form a peer-to-peer network, the proportion of lost packets, as well as their delay, is very high, not even mentioning the very irregularly distributed throughput, which is generated. These values, and additionally the whole loss rate of ca. 25% of all packets in the whole network, almost demonstrate

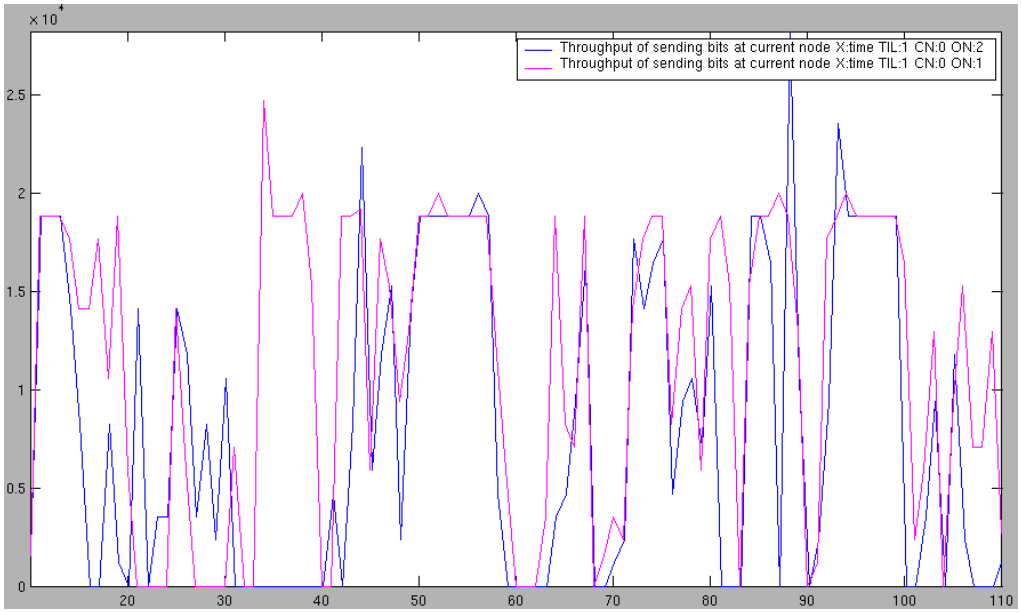

Figure 5.6: Throughput Rudimentary Topology from Server to Clients

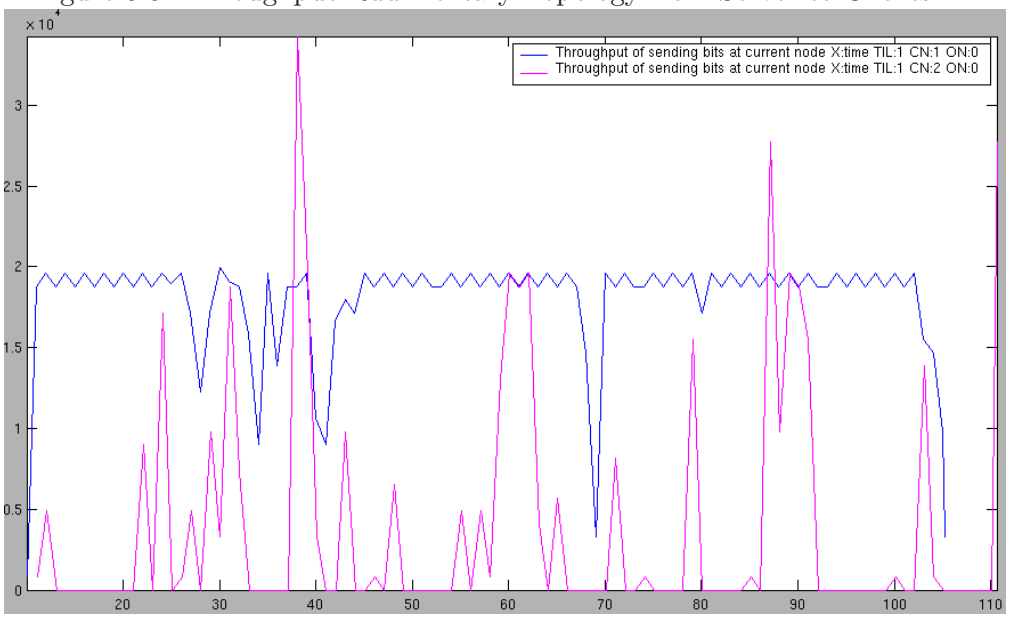

Figure 5.7: Throughput Rudimentary Topology from Clients to Server

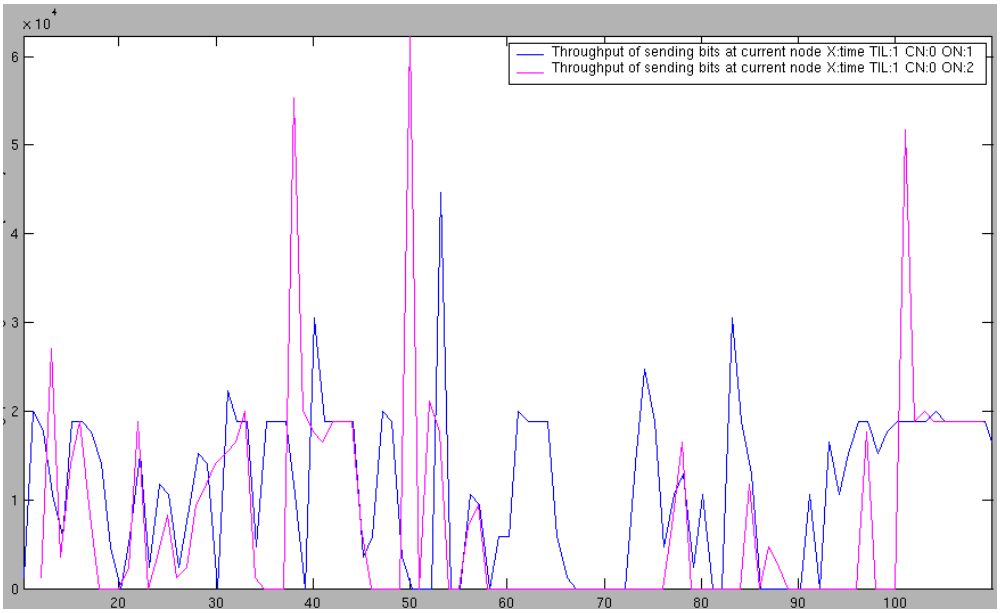

Figure 5.8: Throughput BitTorrent Topology from Server to Clients

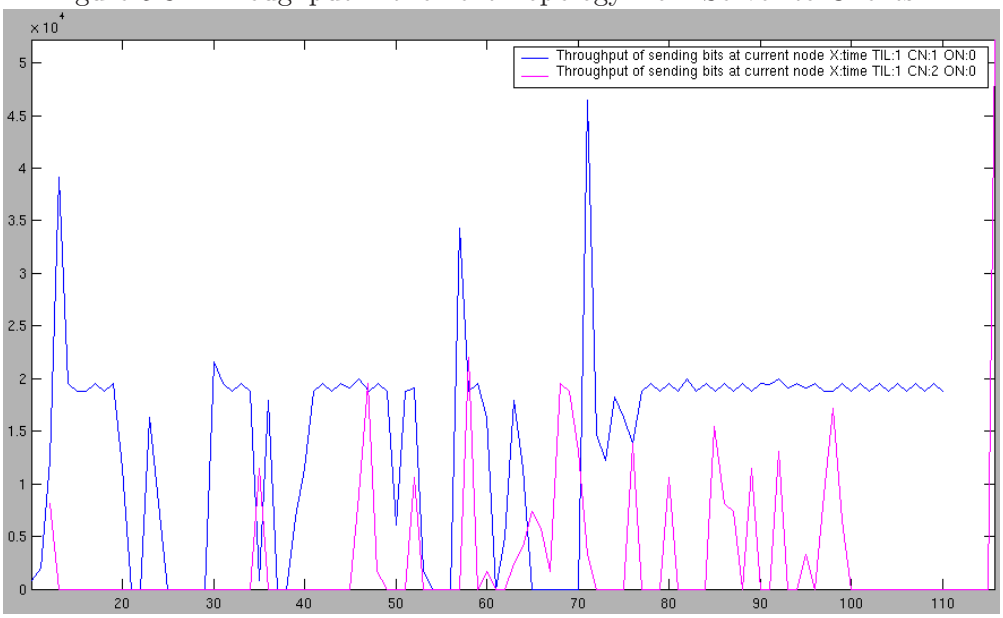

Figure 5.9: Throughput BitTorrent Topology from Clients to Server

a close collapse.

Topology Data Rate with 54Mbs

As at the beginning of this section described, both analyses are based on the same kind of scenarios, differing only in the data rate. As expected, using a higher data rate of 54Mbs, many more packets were sent, partly almost twice the amount, refer to table 5.4.3. However, the "Rudimentary Topology" sent 25.58% fewer packets than the "BitTorrent Topology". Also the number of lost packets in the "Rudimentary Topology" is increased, showing a higher loss of up to 73.80%.

However, what seems surprising is the fact that, in spite of the higher number of successfully delivered packets, also in the "BitTorrent Topology" no decent exchange of data between the server and the clients appears in general. Therefore, playing a computer game becomes impossible, excluding the connection between Server and Client 1, because the delay is higher than the RTT 100-150 msec of a playable course, which is between 100 and 150 ms, see section 5.1.2.

	Ser -> Cl_1	Ser -> Cl_2	Cl_1 ->Ser	Cl_2 -> Ser
Rudimentary Traffic				
Send	1171.3 pk	1173.5 pk	2328.3 pk	1667.0 pk
Lost	41.1 pk	73.6 pk	46.5 pk	133.6 pk
LostSend	3.50%	6.27%	1.99%	8.01%
Delay	0.157 sec	0.670 sec	0.795 sec	0.874 sec
BitTorrent Traffic				
Send	1574.7 pk	1560.7 pk	2340.6 pk	2033.8 pk
Lost	11.4 pk	34.8 pk	27.1 pk	35.0 pk
LostSend	0.72%	2.22%	1.15%	1.72%
Delay	0.047 sec	0.186 sec	0.147 sec	0.216sec
Differences (RTBT)				
Send	+25.58%	+24.80%	0.00%	+18.02%
Lost	-72.26%	-52.71%	-41.72%	-73.80%

Table 5.4.3: Comparison of the average values of the playing nodes with 54Mbs data rate

When comparing all packets of the scenarios, the Packets of the three base nodes and the routing traffic, see table 5.4.3, it becomes obvious why the absolute number of packet loss in the "Rudimentary Topology" is much higher than in the other topology. In the "BitTorrent Topology", less than half of the packet numbers had to be send in the BitTorrent Topology and so less than half were dropped or got lost. However, the ratio between sent and dropped/lost packets remains almost equal, presenting 5.78% in the "Rudimentary Topology" and 5.84% in the "BitTorrent Topology".

	Rudimentary Traffic	BitTorrent Traffic
Send	46609.2 pk	20446.5 pk
Dropped	2393.5 pk	1035.6 pk
Lost	301.3 pk	160.4 pk
Dropped and Lost	2694.8 pk	1196.0 pk
DL/Send	5.78%	5.84%

Table 5.4.3: Comparison of the packets in the whole scenario

The deviations of the sent and lost packets between the single scenarios, lookup Table 5.4.3, of one category were clearly higher in the BitTorrent topology compared to the rudimentary traffic, which achieved a more even traffic during these scenarios. This is due to the number of clients, whose connections increases exponential, in BitTorrent, whereas in the rudimentary traffic they form a mean value. Although the average number of connections is equal, BitTorrent can obtain substantially better results, because it also can produce scenarios, which have fewer connections. That allows the conclusion that the "Rudimentary Traffic Topology" is not quite able to imitate such a complex topology like the BitTorrent. The Table , lookup Table 5.4.3 shows the absolut lost packages from scenario 1 to 10, in the rudimentary topology the maximal difference between the lost packets are 993, whereat the maximal difference of the BitTorrent topology is up to 2555 packets.

	Rudimentary Traffic	BitTorrent Traffic
1. Szenario: Lost Packages	2345 pk	1184 pk
2. Szenario: Lost Packages	2123 pk	1753 pk
3. Szenario: Lost Packages	2605 pk	2709 pk
4. Szenario: Lost Packages	3116 pk	1269 pk
5. Szenario: Lost Packages	2680 pk	698 pk
6. Szenario: Lost Packages	2924 pk	964 pk
7. Szenario: Lost Packages	2651 pk	331 pk
8. Szenario: Lost Packages	2965 pk	1449 pk
9. Szenario: Lost Packages	2587 pk	1449 pk
10. Szenario: Lost Packages	2952 pk	154 pk
Maximal difference	993 pk	2555 pk

Table 5.4.3: Comparison of the whole packet loss in all scenarios

This time, for the comparison of the scenarios, the scenario 5 of the "Rudimentary Topology" and the scenario 1 of the "BitTorrent Topology" come closest to the average of all constructed scenarios. The rudimentary topology possessed more than $n = 9$ connections and BitTorrent showed $n = 5$, $(n-1)^2 = 16$ connections. When using a data rate of 54MBs, the throughput was much more regular than during the previous analysis. Therefore, the connection of the server to a client and back can be pictured graphically.

Figure 5.10 reproduces the throughput of the rudimentary topology, which demonstrates to be much less regular and uniform than in the BitTorrent topology, although less connections exist. Especially during the connection between the server and client 2, the throughput often decreases under $1.9 * 10^4$ bits, and in the time interval between the 18th and the 40th second this connection does not even have any throughput at all.

In contrast, the throughput of the "BitTorrent Topology" is especially uneven from the beginning to the 35th second. But after that time the throughput normalizes to a relatively uniform level, revealing hardly any differences in the sending behaviour of the single nodes, apart from a quite uneven phase from the 83rd to the 90th second.

The same characteristics are reflected in the analysis of the jitter. Figure 5.12 und 5.13 demonstrate the jitter from the server to the client and vice versa, during the rudimentary topology. The same is demonstrated for the BitTorrent topology by the figures 5.14 und 5.15.

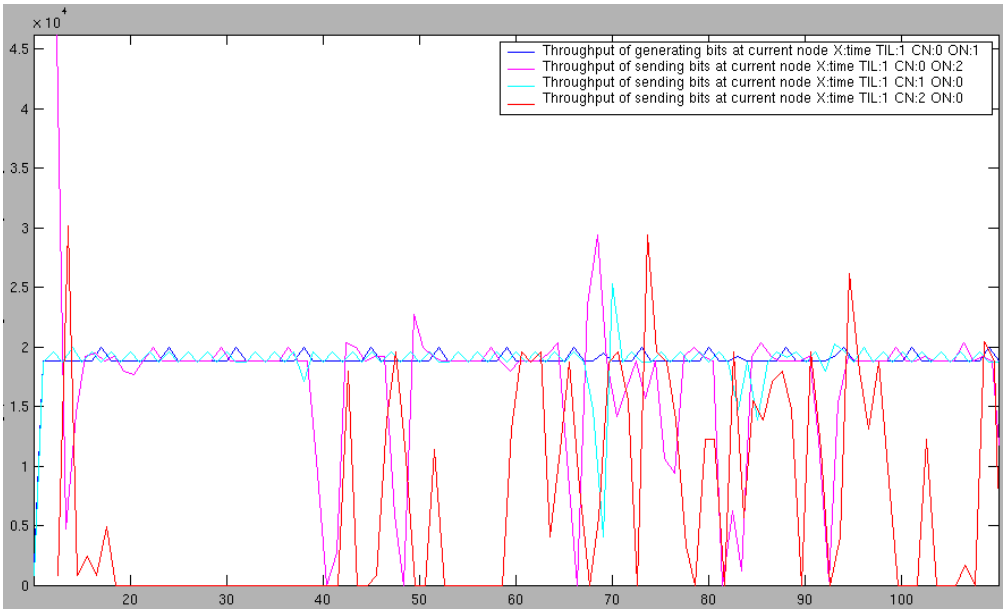

Figure 5.10: Throughput Rudimentary Topology from Server to Clients and back

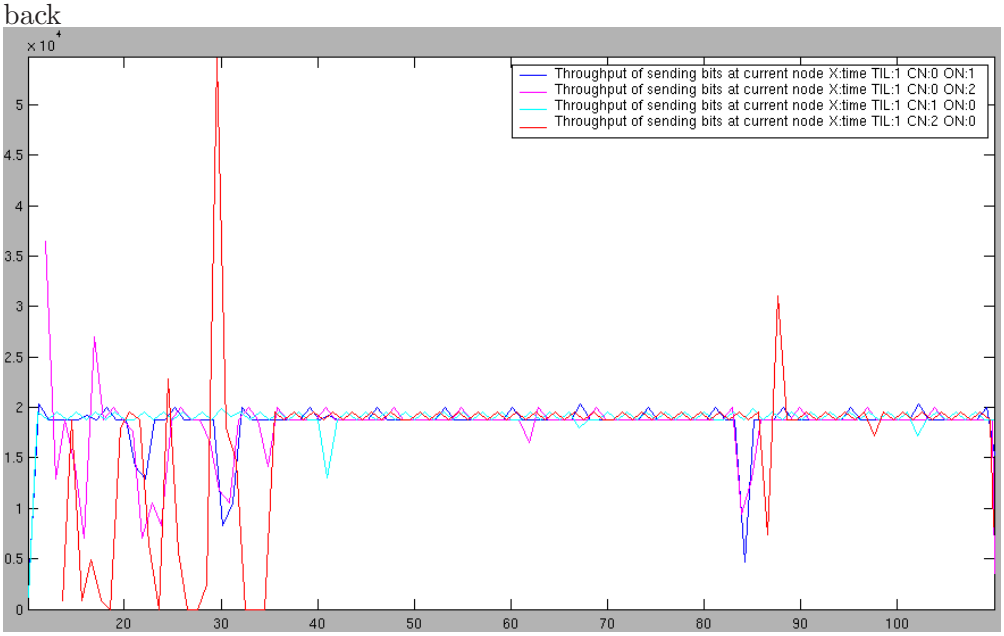

Figure 5.11: Throughput BitTorrent Topology from Clients to Server and back

What was already demonstrated by the previous analysis of the throughput is also transferable to the jitter. In the rudimentary topology the jitter times are very uneven during the entire simulation time. Again, especially the connection between client 2 and the server reveals particularly high jitter times, whereas the connection of client 1 shows shorter and more regular jitter times.
In contrast, the BitTorrent topology demonstrates significantly higher jitter times, especially at the beginning, which then normalises during the course of simulation, apart from a few peaks. Especially this behaviour indicates that the average delay times, determined at the beginning of table 5.4.3, have to be judged very carefully.

That is why we have to look at the delay times of the scenarios once again, choosing a representative scenario. As before, we choose scenario 5 of the rudimentary topology, refer to figure 5.16, and scenario 1 of the BitTorrent Topology, see 5.17. But this time we only look at the bidirectional connections from the server to the client. It reveals clearly that in the rudimentary topology the delay times are so high that no decent exchange of data between the server and client 1 was possible during the whole simulation time. In contrast, in the BitTorrent topology the delay times are low in the time interval of the 52nd to the 83rd second and after the 87th second, so that the wanted exchange of data was able to occur.

In general, we can conclude that the attempt to bring the rudimentary and the BitTorrent topology to a relative comparable standard was successful in the scenarios using a data rate of 2Mbs, but differs in the other scenario with 54Mbs. This big difference of the scenarios is surely based on the fact that the rudimentary topology establishes and closes connections in a random based manner during the whole course of the scenarios, see 5.4.1. In contrast, the BitTorrent Topology establishes connections already at the beginning with a higher probability to prevent an accumulation of connections in the middle of the simulation, see 4.7.3. Additionally, it closes these connections only at the end of the scenarios. Surprisingly, this behaviour causes such an enormous difference between the two topologies, resulting in the effect that the network of the rudimentary topology does not calm down and produces such miserable delay times.
Finally it has been mentioned, that it is not easy to generate such a compli-

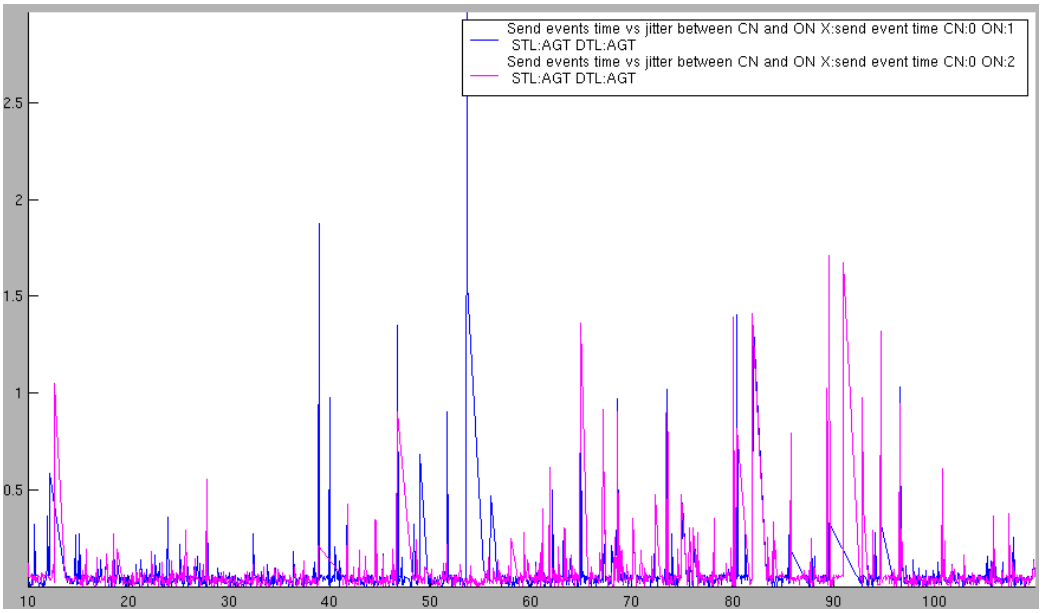

Figure 5.12: Jitter Rudimentary Topology from Server to Clients with 54Mbs

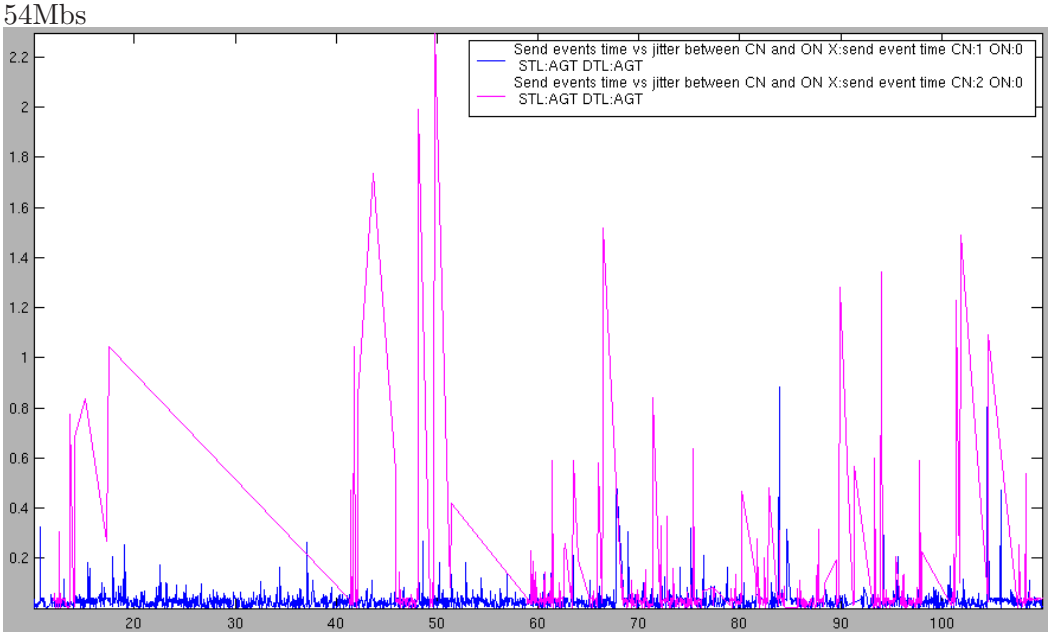

Figure 5.13: Jitter Rudimentary Topology from Clients to Server with 54Mbs

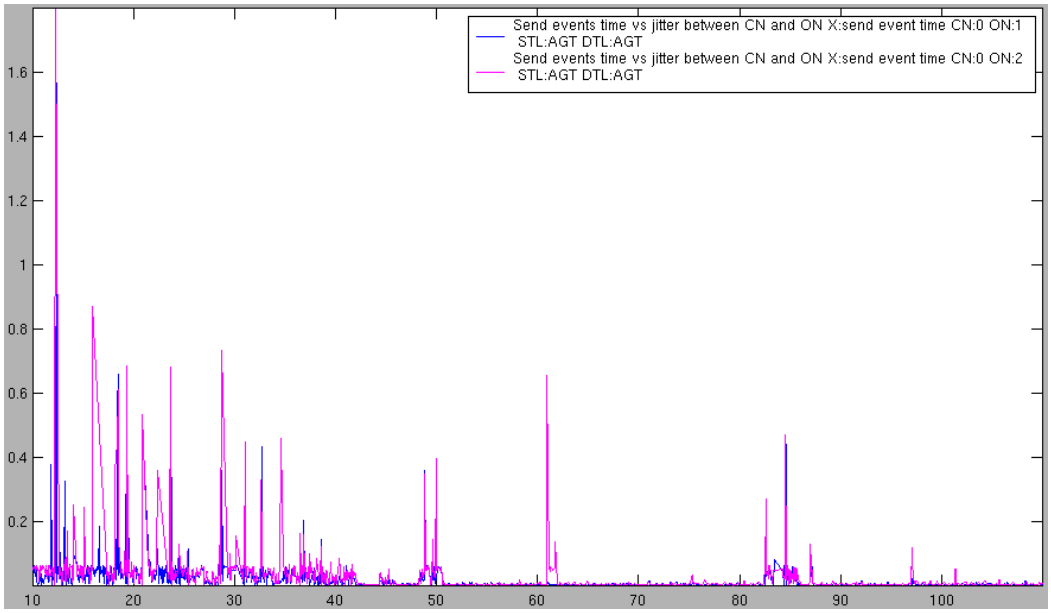

Figure 5.14: Jitter BitTorrent Topology from Server to Clients with 54Mbs

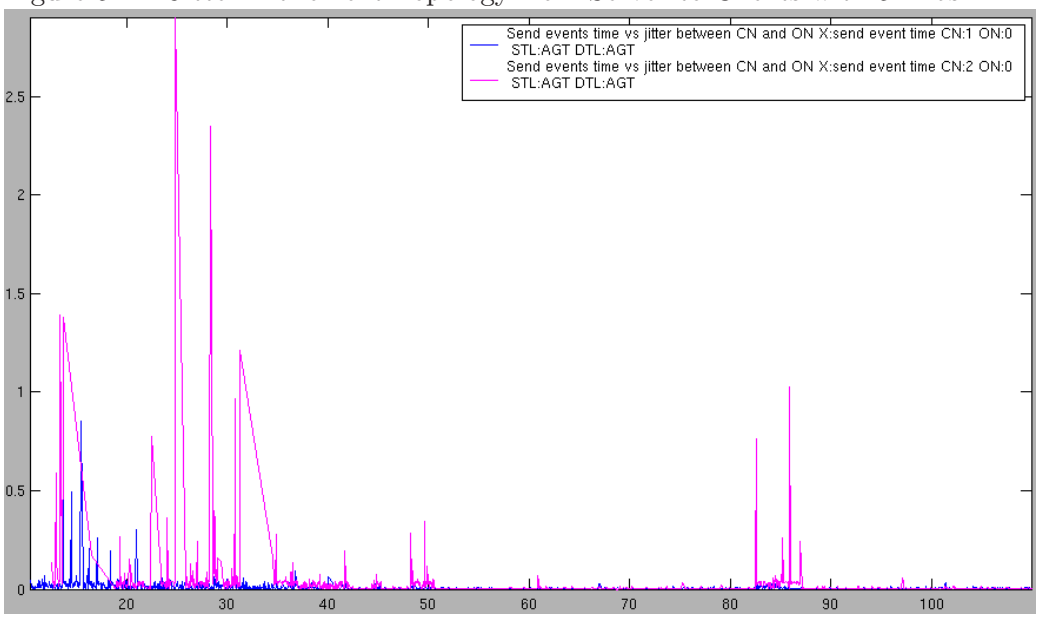

Figure 5.15: Jitter BitTorrent Topology from Clients to Server with 54Mbs

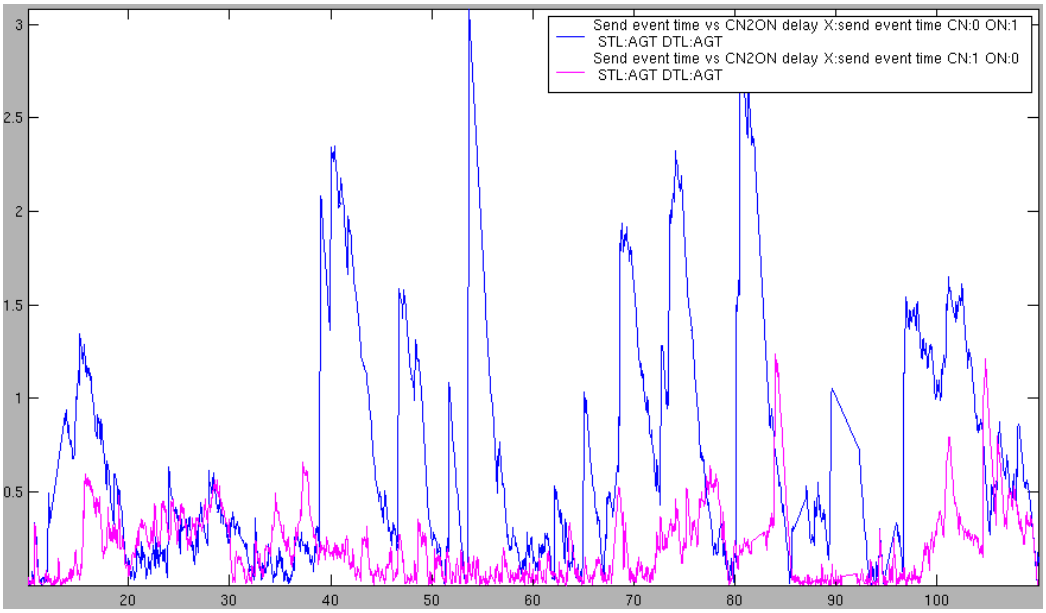

Figure 5.16: Delay Rudimentary Topology from Server to Clients 1 with 54Mbs

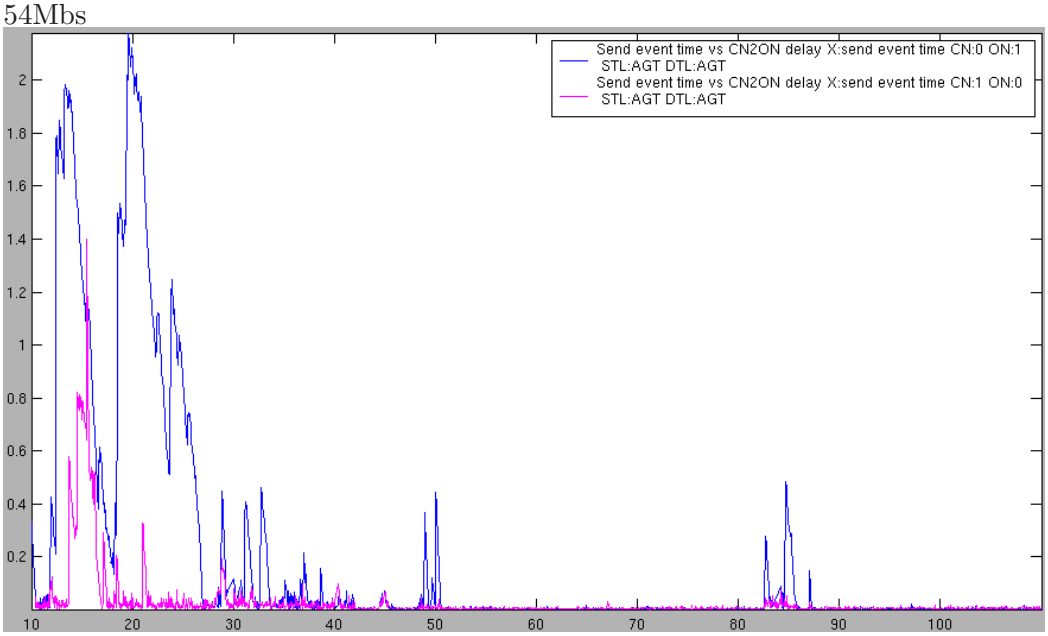

Figure 5.17: Delay BitTorrent Topology from Clients 1 to Server with 54Mbs

cated network as the BitTorrent topology with such a simple plugin as the rudimentary topology, because their behaviours differ a lot from each other. On one hand, it was possible to generate a comparable traffic using a data rate of 2Mbs, but on the other hand, the same scenarios differed enormously, when using a higher data rate, because then, the rudimentary topology is not able to imitate the calm phases, during which a continuous data rate is able to occur.

5.5 Summary

In this chapter we described the planning, the execution and the analysis of the single scenarios, constructed for testing the random plugin categories: Start point, movement and traffic. The scenarios were based on computer playing nodes, which accorded to the "Network Gaming Traffic Modelling" (Literaturliste [9]).
At first we evaluated the random start point category, by comparing the "Bounding Box" random start point algorithm with the "Motion based Pre Calculation" algorithm. The analysis exposed that significant differences between these two methods did not exist. The varying start positions of the nodes did apparently not influence the following course of events in the network.
The comparison of the mobility motion algorithm revealed that, in general, the "Gauss Markov" did score clearly better than the "Random Waypoint". This was probably due to fact, that the nodes, generated by "Gauss Markov", moved less linear, and therefore, they changed their positions slower, resulting in a declined fluctuation of this topology. This behaviour can surely be enhanced or attenuated by means of more restrictive configurations. This means, that the "Gauss Markov" is to be preferred to the "Random Waypoint", revealing a higher level of variations.
Finally, we compared both existing traffic plugins of the NS-Mapper with each other. By means of probability calculations we tried to make the "BitTorrent Topology" and the "Rudimentary Topology" comparable. This comparison was contradictory, depending on the chosen data rate. The analysis, using a data rate of 2Mbs, showed that the topologies exhibited a very similar behaviour. However, this was not confirmed, using a higher data

rate of 54MBs. Hereby, with the “Rudimentary Traffic”, the establishing and the closing of connections during the simulation time produced so much overhead, that calm phases could not be achieved.

Chapter 6

Conclusions and Outlook

This chapter will provide a summary of the performed work and a presentation of the obtained results, including critical comments. Finally, it will present an outlook of the features, which the NS-Mapper might expect in the near future.

6.1 Conclusion

In this diploma thesis, at the beginning of chapter 2, we presented an overview of the possible, worth-considering tools, which deal with the random based construction of automatic generated scenarios. We concluded, that the selection of software, dealing with this topic is very limited and additionally, fails to achieve a user-friendliness, regarding the merging of different methods within these programs. We presented a set of random based algorithms, which were able to expand the realistic behaviour of mobile nodes in the NS-Mapper scenario editor. In detail, this contained the categories of the random based start point, the automated motion and the traffic. As far as a balanced selection was possible, algorithms were chosen, which seemed to provide a realistic behaviour, and which covered a large range of possible applications or where already commonly used, respectively. However, the lack of existing evidence studies evaluating this field, proofed to be a problematic. That is why we ourselves had to compile known algorithms and had to evaluate them using special criteria, such as comprehensible natural behaviour, their realizability to function with respect to the pre-determined structure of the NS-Mapper or their current frequency of usage. After a comprehensive evaluation, we chose the best suited method for each of the three

categories, start point, motion and traffic. For the category of the random start point we decided for the "Motion Based Pre-Calculation" mechanism, because it is able to place the start position of a mobile node in a way that usually is only obtained during the course of simulation. For the random category "Mobility Motion" we chose the "Gauss Markov" algorithm, because it includes the last direction and velocity of mobile nodes in its following calculations, by referring to a rudimentary memory of every movement section. Therefore, the nodes do not move completely random based. This seems to be most suitable for imitating the behaviour of human beings. As already mentioned previously, unfortunately, especially regarding the movement behaviour no evidence studies exist or at least, are available. In spite of this disadvantage, the "Gauss Markov" algorithm seemed to come closest to a natural behaviour, as it might appear during strolls or when shopping. Finally, for the traffic model we chose the "BitTorrentTopology", because, on one hand, it is a frequently used protocol in the internet, possessing a high number of users, and on the other hand, because its complex topology is not easy to construct and to simulate, in contrast to alternative models and therefore, providing an adequate challenge. In chapter Design Concepts 3), we described at the beginning how to use dynamically loaded codes in unchangeable Jar packages. In this case, the codes are the integrated random plugins. Because the integration of program parts is not possible during the run time of the Jar packages, the program has to administrate both, a virtual and a real data system. Thereby, all plugins can now be newly acquired if needed, and if necessary, outdated program parts can be exchanged in a running status. Subsequently, we dealt with the access to the data structure, and with the question which methods suited best for providing all information, which are necessary for the execution of a plugin. We decided for a static data structure and not for a RMI administrated structure. Although a RMI-administrated data structure guarantees a higher level of safety and control, the expenses for administrating this structure would have been immense, because, for instance, objects loose their references to all previously attached data structures, when getting transmitted. Later we explained how to construct plugins in Java in general, describing how to load, execute and administrate dynamic code during the runtime. Additionally, we explained the internal construction of a plugin, what interfaces we implemented, and what guidelines we integrated for the plugin developer to ensure

a regulated course of events and a simple and structured implementation: Roughly speaking, the plugin passes three hierarchic sequences: the init, run and finish or abort method. The init method, for example, decides if a plugin is allowed to be executed in regard to the current condition. The run method contains all functionalities of the plugin, for example a calculation of the node placement. The finish or abort method, which complete the calculation cycle, depending if the run method was successful. Additionally, we tried to guarantee a high reusability when constructing the plugin, ensuring that the once realized concepts are reusable for other plugins and furthermore, guarantee a simple exchange of data within the whole program, including the exchange between the plugins themselves.
In the chapter Implementation 4, we tried to present the most important parts of the newly developed concepts. Because of their complexity, it was not possible to explain all achieved improvements. Therefore, we concentrated on the parts, which were necessary for the development of a random based plugin. Additionally, we outlined the hierarchy of the data structure, explaining how it is functionally constructed and how to invoke the single parts of it. We explained the parts of the framework, which are supposed to simplify the development of future plugins and the processing of data for the developer. We described how to create interactions with the user by means of the PaintingArea, marked objects, as well as mouse and keyboard events. We explained how to make, prepare and invoke data in a plugin by means of the standardized OptionsKeeper and how to exchange data between the single plugins. Furthermore, we defined the PathLine class, which performs all calculations of the movement of nodes. This class ensures that the paths of the mobile nodes, in dependence on the changing and limited environment topology, are divided in single segments and are corrected, if necessary. The PathLine class provides that nodes pass from one MovementField to another, in respect to the timing. Finally we explained the implementation of the single random plugins, which were developed in this diploma thesis. We specified the configuration possibilities of the start point plugin, of the movement and traffic plugin, and showed how each plugin functionally works in the NS-Mapper. We described in detail which adjustments had to be performed to make the plugins function in the limited conditions of the NS-Mapper. Especially the Gauss Markov plugin had to be extended strongly for being able to be properly integrated in the NS-Mapper. There-

fore, new calculations had to be integrated in this algorithm, as for example, the distance calculation of borders of an uneven polygon, as well as calculations of the angles, so that the repulsion direction of any border could be calculated individually. These calculations influenced the complexity of this algorithm decisively. Also the BitTorrent traffic plugin had to be adjusted, so that it was able to function with the rudimentary conditions of any simulator. For that, the connections to a tracker and the connections between the clients had to be simulated, although the used simulator did not possess a direct implementation of the BitTorrent protocol.

In chapter Evaluation 5 we evaluated the single programmed plugins by comparing them to former used plugins to see if significant changes during the simulation in the NS-2 arose. Hereby, it had to be considered that constructions had to be chosen, that guaranteed a comparability of the generated scenarios. At first we defined a basis for the scenarios, which was maintained in all simulated scenarios. This basis accorded with the "Network Gaming Traffic Modeling" Technical Paper[9] and contained three static nodes, which were not able to make contact with each other in a direct way. One node functioned as the game server, whereas two clients were able to connect to it bidirectionally. Additionally, we generated 20 further nodes, which were evaluated in regard to the requirements of the topology. It was analysed how they influenced the connection between the three static nodes. To be able to exclude all disruptive factors, the scenarios were kept as simple as possible.

In the analysis of the random start point environment, lookup section 5.2, it was revealed that no significant difference appeared between distributing the start point of mobile nodes evenly in the scenario or distributing them random based when calculated by a mobility model. The average values of all constructed scenarios were very similar in the comparison of these two ways to define start points.

For the analysis of the random mobility model environment, lookup section 5.3, the nodes were subsequently either moved by the "Random Waypoint" plugin or by the "Gauss Markov" plugin. The difference between these two mobility models was obvious, still small. However, it has to be regarded that the traffic only occurred between the three computer game playing nodes, whereas all further nodes were used for the routing. Especially on that condition it is surprising, that the "Gauss Markov" scored in all scenarios better

than the “Random WayPoint” algorithm. Most probable this is due to the fact that the mobile nodes, moved by the Gauss Markov, remained longer in a specific area and therefore, the general topology changed slower, which again influenced the number of lost packets. Even the delay was averagely better, achieved by the Gauss Markov. For the analysis of the random traffic environment, lookup section 5.4, the two traffic plugins of the NS-Mapper were compared with each other, consisting of “Rudimentary Topology” and the “BitTorrent Topology” plugin. In this scenario we firstly had to generate a kind of traffic, which made the simulations comparable. Therefore, we had to perform calculations, which originated from the probability theory. Then we simulated for both random traffic algorithms the same scenarios, once using a data rate of 2Mbs and once with a data rate of 54Mbs. The analysis of the scenarios with a data rate 2Mbs seemed to indicate that the two compared topologies did not differ from each other. However, this was most probably due to the fact that the generated traffic forced both topologies to use all of its capacities. Thereby, at the end, the three computer game playing nodes had lost almost half of their packets and the delay between their connections showed to be 1-2 seconds, which definitely makes a decent game impossible. When evaluating the topologies with a data rate of 54Mbs, the differences between the rudimentary traffic and the BitTorrent traffic were very apparent. The BitTorrent topology scored better than the rudimentary topology in all calculations. The number of sent packets was up to 1/4 higher and the lost packets even 1/2 lower than with the rudimentary topology. Additionally, the delay times in the BitTorrent topology were only between the server to client 1 quite acceptable, that they would have been able to perform a computer game; however, the connection between clien1 to server and the connections between server and client 2 and back still remained above the possible delay times of a computer game. The deviations of the sent and lost packets between the single scenarios of one category were clearly higher in the BitTorrent topology compared to the rudimentary traffic, which achieved a more even traffic during these scenarios. This is due to the number of clients, whose connections increases exponential, in BitTorrent, whereas in the rudimentary traffic they form a mean value. Although the average number of connections is equal, BitTorrent can obtain substantially better results, because it also can produce scenarios, which have fewer connections. That allows the conclusion that the “Rudimentary

Traffic Topology" is not able to imitate such a complex topology like the BitTorrent.

At the beginning, we explained that we focus on creating the NS-Mapper as a scenario editor, which is able to imitate the natural behaviour of mobile nodes as realistic as possible. This can be achieved when the NS-Mapper is able to combine and merge many different start point, motion and traffic strategies within one single scenario. Surely, we now can claim that differences exist between the single categories, as described in chapter 5, however, by means of our data we can in the end not decide confidently if the implemented strategies and techniques are able to imitate the reality. This is due to not existing evidence studies concerning the natural distribution of start positions and the movement behaviour of real human beings. Without scientifically sustained data, the natural behaviour of implemented algorithms can be supposed, but definitely not be proofed.
The NS-Mapper has developed into a simple-to-use modeller for Ad-hoc scenarios. It demonstrated, while constructing the scenarios, that the realization of scenarios by means of this software, did only demand a fraction of the time, which was necessary for the planning and the analysis. When also considered that a user does not have to be trained in such a complex simulator, as for example the NS-2, and additionally, is not forced anymore to merge the different strategies manually, the NS-Mapper provides a valuable tool, which dramatically increases the effort when constructing scenarios.

6.2 Outlook

The NS-Mapper scenario editor is committed to the task to easily plan, construct and test Ad-Hoc networks. It would be a further big step if it could also take over the analysis of trace files. Actually, it already contains the control and the knowledge of all mobile nodes and their connections. Therefore, it would be a logical expansion to simply mark nodes, to analyse them if trace files exist for it. Every user, who analysis trace files, usually develops his own analysing tools, which definitely is a waste of time. The only comprehensive software, which is currently available and which also was used during this diploma thesis, is TraceGraph. Even if this tool is able to perform a large amount of calculations (as for instance, calculation of

send, dropped and lost packages, delay, throughput, jitter and RTTs), this software did not function very convincing, when making our evaluations. The calculations took a lot of time and the automatic scripting function for analysing a big number of scenarios could not be get going. Additionally, the development of this tool remains "Closed Source".

Concerning the implementation of the internal features, the OptionKeeper, which administrates the options in the NS-Mapper, should be revised, for being able to work controlled on some values, as for example minima und maxima. Once revised, the developer of the plugin could invest this time saving in more important things than for the correction of errors of the single values.

As already discussed, the data structure for the exchange of relevant values should be replaced by a RMI structure, which would offer a better level of security for the data structure.

Appendix A

Used Acronyms and Shortcuts

Acronym and Shortcuts	Description
AC	Average Connections
ACP	Average Connection Probability
AODV	Ad-hoc On-demand Distance Vector
AT	Average Traffic
BBR	Bounding Box Random
CBR	Constant Bitrate
Cl	Client
DL	Dropped and Lost
DSDV	Destination-Sequenced Distance Vector routing
DSR	Dynamic Source Routing
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
MBPC	Motion Based Pre Calculation
MC	Maximum Connections
MT	Maximum Traffic
pk	packets
RMI	Remote Method Inokation
RTT	Round Trip Time
Ser	Server
TCP	Transmission Control Protocol
TIL	Time Interval Length (1 sec)
TORA	Temporally-Ordered Routing Algorithm
UDP	User Datagram Protocol

Appendix B

CD Contents

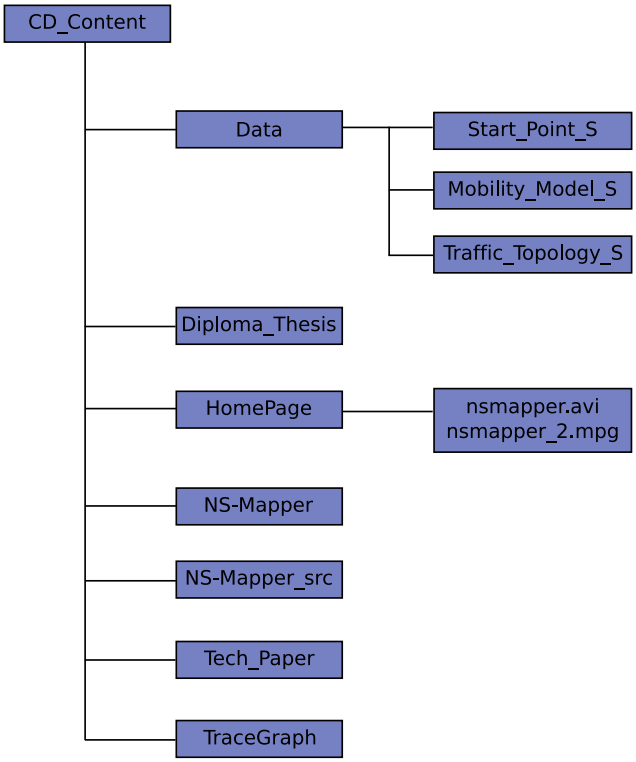

Figure B.1: CD Contend

Figure B.1 shows the DVD content of the enclosed DVD. In the **Data** directory are all calculated szenarions, which were made for this thesis. They are splitted in thre subdirectories, which are **Start_Point_S** for the start point szenarios, **Mobility_Motion** with the movement scenarios and **Traf-**

fic_Topology for the traffic scenarios. The analyse data are made with the TraceGraph software, all calculations of this software have the fileending: mat. All trace and mat files can be processed with TraceGraph. There are also some additional shell scripts. The shell script **dropped.sh** calculates the dropped packeges of all scenarios and the **delay.sh**. Additionally there can appear shell scripts like **dropped_2.sh** and **delay_2.sh** which excludes a specific trace file from the calculations by giving the file number to the end of the script.
In the **Diploma_Thesis** directory are all latex files of the diploma thesis, also with the pdf and ps files.
The **Home_Page** includes the home page of the NS-Mapper. Additionally there are 2 short videos, which show the NS-Mapper at work, they called **nsmapper.avi** and **nsmapper_2.mpg**. The directories **NS-Mapper** and **NS-Mapper_src** keep the NS-Mapper in a Jar file and its source codes. The directory **Tech_Papers** keeps the most papers which are used in the diploma thesis. The directory **TraceGraph** keeps the TraceGraph tool, which was used to analysing the tracefiles of this diploma thesis.

Appendix C

Installing the NS-Mapper

At first you have to get the NS-Mapper tar file and unpack it into your favorite directory. Then download the xstream Jar package from the below link. Copy the xstream package to the NS-Mapper directory and execute:

```
jar i NSMapper[Version].jar xsream[Version].jar
```

This adds xstream to the dependencies of the NS-Mapper. Then start the NS-Mapper with:

```
java -jar NS-Mapper[Version].jar
```

Index

Bibliography

[1] C. Bettstetter. Mobility modeling in wireless networks: Categorization, smooth movement, and border effects. Technical report, Technische Universität München, 2002.

[2] C. Chiang. Wireless network multicasting. Technical report, University of California, 1998.

[3] M. G. Christian de Waal. Bonnmotion. Institut für Informatik der Universität Bonn. http://web.informatik.uni-bonn.de/IV/Mitarbeiter/dewaal/BonnMotion/.

[4] T. C. J. B. V. Davies. *A Survey of Mobility Models for Ad Hoc Network Research. Wireless Communications & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*. 2002.

[5] V. Davies. Evaluating mobility models within an ad-hoc network. Technical report, University of Colorado, Master Thesis, 2005.

[6] dev radio. Java - evolution einer sprache. Radio Show. http://www.ulm.ccc.de/dev/radio/detail?id=45.

[7] Diverse. *The Network Simulator ns-2 Documentation*, 2005.

[8] E. Eden. Evaluierung von routingprotokollen bezueglich des latenzverhaltens in mobilen ad-hoc netzwerken. Technical report, TU-Braunschweig, 2005.

[9] J. Faerber. Network gaming traffic modelling. Technical report, 2002.

[10] L. Frenzel. Neutral im sinne der qualität. Eclipse Magazin, 2006. http://eclipse-magazin.de/itr/online_artikel/psecom,id,778,nodeid,230.html.

[11] B. L. Z. Haas. *Predictive Distance-Based Mobility Management for PCS Networks. In: Proceedings of IEEE Infocom Conference. New York, NY, USA :IEEE,.* 1999.

[12] Z. Haas. *A new Routing Protocoll for the reconfigurable wireless Networks In: Proceedings of the IEEE International Conference on Univeral Personal Communications, IEEE.* 1997.

[13] D.-I. H. Hellbrück. Ansim. University of Lübeck.
http://www.ansim.info/.

[14] M. W. K. Jeffay. Packmime. Lucent technologies, University of North Carolina at Chapel Hill.
http://dirt.cs.unc.edu/packmime/.

[15] T. I. H. Korth. Dynamic source routing in ad-hoc wireless networks. in mobile computing. Technical report, Kluwer Academic Publisher, 1996.

[16] R. M.-S. Moser. *The Analysis of the Optimum node density for ad hoc mobile networks. In Proceedings of the IEEE International Conference on Communications (ICC).* 2001.

[17] S. Northcutt. Bittorrent considered harmful to intellectual property. SANS Technology Institute.
http://www.sans.edu/resources/securitylab/227.php.

[18] B. Roemer. Bonntraffic. University of Bonn.
http://web.informatik.uni-bonn.de/IV/bomonet/BonnTraffic.htm.

[19] I. Stepanov. Canumobisim. Universität Stuttgart.
http://canu.informatik.uni-stuttgart.de/mobisim/.

[20] Sun. Java 2 platform standard edition 5.0 api specification. java.lang.reflect.
http://java.sun.com/j2se/1.5.0/docs/api/index.html.

[21] J. M. G. L. D. T. E. Sykas. Mobility modeling in third-generation mobile telecommunications systems. Technical report, 1997.

[22] tomh. Web traffic generator.
http://www.tomh.org/software/.

[23] unknown. Usc mobility generator tool.
http://nile.usc.edu/important/software.htm.

[24] unknown. Video traffic generator. Carleton University.
http://www.sce.carleton.ca/amatrawy/mpeg4/mpeg4_traffic_README.

[25] Various. Eclipse platform api specification. Eclipse Community.
http://help.eclipse.org/help32/nftopic/org.eclipse.platform.doc.isv/reference/api/index.html.

[26] Various. Glomosim. Ucla Parallel Computing Labroratory.
http://pcl.cs.ucla.edu/projects/glomosim/.

[27] L. P. O. L. K. Wehrle. Modulares mobilität framework für netzwerk simulationen. Technical report, Universität Tübingen, 2005.

[28] E. A. Öeznur Öezkasap. A classification and performance comparison of mobility models for ad hoc networks. Technical report, Koc University, Department of Computer Engineering, 2006.