ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Diploma Thesis

ManS

A Simulation Framework for Mobile Ad-Hoc Networks

Roger Kehrer
roger.kehrer@gmail.com

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisor: Roland Flury

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zürich
Spring 2006

Contents

1 Introduction **5**
- 1.1 Motivation . . . 5
- 1.2 Assignment . . . 5
- 1.3 Overview . . . 6

2 Related Work **7**
- 2.1 NS2 . . . 7
- 2.2 GloMoSim . . . 7
- 2.3 SanS . . . 8
- 2.4 ManS . . . 8

3 Basic Decisions **9**
- 3.1 Synchronous vs. Asynchronous Simulation . . . 9
- 3.2 Simulation Process . . . 10
- 3.3 Models . . . 10
- 3.4 NodeCollection . . . 12
- 3.5 Projects . . . 12

4 Design and Implementation **15**
- 4.1 Runtime System . . . 15
- 4.2 Synchronous Simulation . . . 15
- 4.3 Interfaces of the Models . . . 18
- 4.4 SmartNodeCollection . . . 20
- 4.5 Configuration . . . 22
- 4.6 Additional Tools . . . 23
  - 4.6.1 Data Series . . . 24
  - 4.6.2 Logging . . . 24
  - 4.6.3 Distributions . . . 24

5 Results **25**
- 5.1 GUI Mode . . . 25
- 5.2 Batch Mode . . . 27
- 5.3 Performance . . . 29

6 Outlook **31**
- 6.1 Open Problems . . . 31
  - 6.1.1 Garbage Collection . . . 31
- 6.2 Future Work . . . 32

6.2.1 Asynchronous Mode 32
6.2.2 XML-Graph-file 33
6.2.3 Sleep 33
6.2.4 3D 34
6.2.5 Logging 34

7 Conclusion **35**
7.1 The Results 35
7.2 Personal Experience 35

A User Manual **37**
A.1 Installation 37
A.2 First Steps 38
A.2.1 Parameters 38
A.3 Usage 40
A.3.1 Creating Projects 40
A.3.2 Modifying the Configuration 41
A.3.3 Implementing Algorithms 42
A.3.4 Maps 43
A.3.5 Sample Application 44
A.3.6 Tips 47
A.3.7 Tuning 48

Chapter 1

Introduction

1.1 Motivation

A distributed computing environment can often be described with a graph, where the vertices represent the computing devices and the edges represent the connections between them. One first approach to solve a problem on such a graph is to gather all necessary information at a single device, compute the solution, and send it back to the nodes. However, for many problems, we can achieve much better performance if each vertex computes the solution based on its local view.

An algorithm that computes the solution (or an approximation) based on partial information of the problem is called local. Developing and analyzing such local algorithms is a non-trivial task, mostly intuit. Such algorithms are often 'executed' with paper and pencil on small graphs to verify the correctness or understand certain properties. This paperwork is very vulnerable to errors and very slow. Additionally it is not possible to execute huge simulations with 1000s of nodes by hand. In order to facilitate this work, we built this simulation tool.

1.2 Assignment

The task of this thesis is to develop a simulation framework for local algorithms that runs on graphs. We are looking for a generic tool that allows us to quickly implement algorithms and run it on several distinct graphs, rendering unnecessary the tedious and error-prone paper and pencil work. As we want to simulate the algorithms not only on small graphs of about 10 to 100 nodes but also on graphs of the size of 10'000s or even 100'000s of nodes, performance is also an important issue.

To test the algorithms in a more realistic model we want the algorithms not only to run on static graphs but also in a dynamic environment. Another important point is to have an easy way of specifying the graphs and their behavior. This behavior could be the mobility of the nodes, the connectivity of the graph and so on. We want the framework to allow the user to specify the environment in a plug-in-style where one can run the same algorithm with different configurations by combining different behavior models.

1.3 Overview

The rest of this document is structured as follows. In Chapter 2 we present the related work. We also discuss advantages and drawbacks of these solutions. Chapter 3 then shows the decisions that we had to take at the beginning of the work. In the subsequent chapter, the design and important pieces of the implementation are described. Chapter 5 shows the results of the work and in chapter 6, unsolved problems are presented and an outlook on future work is given. In Chapter 7 the work is concluded by summarizing the derived results and giving a personal comment. Finally, there is a short user manual in the Appendix.

Chapter 2

Related Work

In this chapter we present the related work. Needless to say that we are not the first ones to have a need for a simulation of ad-hoc networks. There are lots of simulation tools available. In this chapter we give an overview over the most popular tools already existing. We shortly present their key benefits and drawbacks and why or why not this simulation tool would be suitable for our problems.

2.1 NS2

One of the most popular network simulators available is ns2[6]. It was developed at the Information Science Institute of the University of South California[5]. It provides a wide variety of possibilities to simulate networks including support for the popular iso-osi layer implementations like tcp/ip, http, ftp and lots more. This makes this tool a powerful simulation framework for simulating internet based applications and protocols.

But due to this great variety of possibilities it is not optimized for testing simple algorithms that do not use the common network stack but just have to be able to receive and send messages and react on some local information. What we wanted to do is to test algorithms, like for example routing in an ad-hoc network, on their correctness and their performance. Performance measurements could be for example the message complexity or the time complexity. For this we do not need all the networking layers provided by ns2. The high variety of possibilities provided by ns2 also increases the time it takes to get into the simulation process.

Due to this high variety of possibilities, the scalability suffers noticeable. With ns2 only simulations with 100's or some 1000 nodes are possible. As we want the framework to handle 10000's or even 100000's of nodes, ns2 is not suitable for our purposes.

2.2 GloMoSim

Another well-known simulation tool available is the GloMo Simulation[4] which was developed at the Parallel Computing Laboratory of the University of Los Angeles UCLA[7]. Like the ns2 simulation, this framework supports an iso-osi

layered network stack. This is not needed for our purposes. The simulation is written in c and an extension of it called parsec[8]. This makes the implementation platform dependent and one has to work into the parsec programing language before algorithms can be developed. Additionally GloMoSim is not an open source projects. It is only available for free for educational purposes. Like ns2, GloMoSim also suffers from serious limitations about the number of nodes that the simulation can be run with. All these factors make this simulation tool not fitting our requirements.

2.3 SanS

One solution previously implemented at the Distributed Computing Group of ETH[3] was the Simple Ad hoc Network Simulator shortly called SanS[2]. It was mainly developed for a course at ETH and was written in Java. It allows implementing the behavior of nodes connected by edges. Like the solution previously presented it uses an iso-osi layered approach. Thus it is also not fitting our requirements especially because the simulation is bounded to a very low number of nodes.

2.4 ManS

All the solutions presented above are not fitting our requirements. In the following we shortly present the key benefits of our solution.

In our solution it is possible to simulate a high number of nodes. Simulations with up to 500'000 nodes are possible. It is platform independent as it is written in Java. It also provides a simple and easy understandable structure. This grants that it is easy to learn implementing new algorithms and environmental models. Nevertheless the simulation is very fast and even huge networks of 100'000s of nodes can be simulated. Additional to this it provides a modular structure with easy configurable projects.

Chapter 3

Basic Decisions

The core aspect of the framework is to specify easy to use interfaces that allow the user to quickly write simulations but does not affect the performance of the framework. In this section we present some basic decisions we had to take in order to achieve our goal.

3.1 Synchronous vs. Asynchronous Simulation

At the beginning of the project it was intended to provide a simulation that can simulate in synchronous and asynchronous mode.

Synchronous in this context means that the simulation is round-based. This means that in each simulation round every node in the graph gets waken up and performs a step. This makes the simulation process simple but suffers from the problem that when a node is doing nothing over a long period, performance is wasted as it is waken up anyway. Note that in this report we are talking about rounds and steps. When talking about the synchronous mode, there is a difference between these two expressions. A step is what a node does when it is invoked whereas a round denotes the procedure of iterating over all nodes and perform a step on each node.

In asynchronous simulation, every node is just invoked when it has to do something. This means that every action generates an event in the future which is a reaction on the action. Concretely this means that when a node is sending a message, it calculates the time it takes the message to reach its destination and schedules an event for the destination node at this time. Simulating such a system now just means to process one event after the other.

The benefit of the asynchronous mode is that every node is just invoked when it has something to do and thus there is no waste of performance. But the asynchronous mode also has its drawbacks.

The problem is that the system has to pre-calculate events. If one node sends a message to another one, a messageHasArrived-event has to be inserted in the event queue scheduled at the time the message reached its destination. These events can be calculated relatively easy but others can not. For example, it is not easy to determine whether the connection between the sender and the destination existed during the whole sending process. Given that we consider a dynamic network, this problem can only be solved by introducing breakup and

join events for the connections. These events then have to be pre-calculated for the transmission time of the message. But particularly when there is mobility in the system, the calculation of these events can be extremely expensive and very difficult.

We could solve the above problem by forcing the simulation not to support mobility in asynchronous mode. But also without mobility the performance gain can be very small. The performance gain is only significant if there are always lots of inactive nodes in the system or if the message transmission time is high. With a high message transmission time, there are lots of inactive nodes in the system and thus the profit of simulating in asynchronous mode is high.

Due to these problems and because of the limited time of this thesis, the implementation of the asynchronous mode had to be omitted. We decided to just implement the synchronous mode but tried to specify the interfaces in a way that it should be possible to extend it by the implementation of an asynchronous mode. See Section 6.2.1 for a description of the important points to consider when trying to implement the asynchronous mode.

3.2 Simulation Process

As one of the core aspects of the framework was to provide a simulation for 10'000s or even 100'000s of nodes, performance is a critical aspect. One approach (e.g. taken in SanS) was to start a new Thread for every node. It was clear from the beginning, that this approach was not suitable for our problem, as it would be impossible to start a new thread for every node when there are so many of them. This is because it is only possible to run a few threads concurrently. Additionally, we wanted the results of the simulation to be reproducible and not to depend on the scheduler of the Virtual Machine.

Our approach for the synchronous mode is to store all nodes in a central data structure and traverse them in every round. The nodes then perform their step one after the other and not concurrently. This sequential execution generates problems with the consistency between the nodes. For example if a node performs its step, some of the neighbor nodes may have already performed their step while others have not. If a node tries to get information about its neighbor nodes it does not know if this information is still up-to-date or if it is outdated.

In the asynchronous mode it would be necessary to have a central event queue and have the simulation take one event after the other and execute them. In this case there would be no problem with concurrency, because all the events have a precise execution time and thus the events are scheduled one after the other and not quasi-parallel as in the synchronous mode.

3.3 Models

We wanted the user to be able to configure the simulation environment in a plug-in style. Therefore, we decided to have the environment characterized by multiple models[1], each describing a part of the environment. The concrete implementations of the models are exchangeable. This means that the user can simulate the same algorithm with several configurations by just exchanging the

models.

The models are grouped in the following 6 categories:

- ConnectivityModels
- InterferenceModels
- MobilityModels
- ReliabilityModels
- DistributionModels
- MessageTransmissionModels

In the following, we shortly describe the functions of these categories.

The **ConnectivityModels** are responsible for updating the connections. This means they decide whether two nodes in the graph are connected in the current simulation round or not. Two nodes being connected by a connection can communicate. Note that in our framework connections are always directed. It is possible that a node has a connection to another one but not vice versa. One sample implementation of the ConnectivityModels is the UnitDiskGraph-Model.

The **InterferenceModels** are responsible for the interference in the system. These models define how much a sending node is disturbing the sending process of other nodes. As the calculation of the interference may consume a lot of performance, interference can be switched off to get faster results. One sample model of this group is the SignalToNoise-Model.

The **MobilityModels** are responsible for the mobility of the nodes. They define how the nodes move. As the calculation of the mobility can be very expensive, mobility can be switched off completely. One example for a mobility model is the RandomWayPoint-Model.

The **ReliabilityModels** are responsible for the reliability of the connections. These models specify whether a message sent over a connection reaches its destination or not. They are somehow additional to the connectivity and the interference models. With the reliability models it is possible that a message is dropped even if a connection exists. As an example, the probability that a message reaches its destination could be inversely proportional to the distance between the sender and the target. Or a simplified signal-to-interference model or an additional link-level failure can be implemented.

The **DistributionModels** are responsible for the distribution of the nodes on the field and are used when generating a graph. For example the nodes can be distributed randomly over the field. Do not mistake the distribution models for the mobility models. The distribution models specify the *initial* distribution whereas the mobility models specify the movements of the nodes.

The **MessageTransmissionModels** are responsible for the time it takes a message to be transmitted from its origin to its destination. Normally this

will be a constant amount of time. But it is also possible to implement a model, where the transmission time is non-constant. One example of such a non-constant model would be that the transmission time is dependent of the distance between the sender and the destination.

All these models together are characterizing the simulation environment and they can be exchanged to simulate an algorithm under several conditions. This means that you can specify an algorithm and simulate it for example once with the UnitDiskGraph and once with the QuasiUnitDiskGraph.

The ConnectivityModels, the InterferenceModels, the MobilityModels and the ReliabilityModels are the four main models. A separate instance of each of them is stored on each node. On the other hand, the DistributionModels and the MessageTransmissionModels are the minor models. There is only one global MessageTransmissionModel for all nodes in the simulation and the DistributionModel is only used to generate the initial node deployment.

See 4.3 for the specification of the interfaces of the models.

3.4 NodeCollection

All nodes are stored in a central data structure and are traversed in every round to execute each nodes step. This central data structure is called NodeCollection. In the following we are describing the properties it has to guarantee.

When calculating the connections between the node-pairs, it is important not to check all the pairs in the graph. Checking all the pairs would lead to a complexity of $O(n^2)$ tests. To increase the performance, we wanted to just check the *possible* connections. For example surveying the UnitDiskGraph-Model, it is not necessary to check all nodes pairs for a connection but only the ones having a distance smaller than the radius of the unit disk. The NodeCollection allows to retrieve a set of possible neighbors. When updating the connections, the connectivity model can get for each node a list of *potential* neighbors, i.e. a subset of nodes which are close to the node. This mechanism reduces the cost of updating the connections significantly.

Additionally, it also has to be possible to traverse all nodes in an efficient way. Therefore, the datastructure has to grant a fast traversal of the nodes and an efficient neighbor search. See Section 4.4 for a description of the concrete implementation of the NodeCollection.

3.5 Projects

Using the framework, users will write code to implement new models and node behaviors. There are some sample implementations of nodes and models included in the framework. If every user would include his newly created classes in the root folder structure of the framework, the amount of code would grow higher and higher. As we didn't want the size of the framework to grow with every new user we introduced the concept of projects.

A project is in fact just a folder structure with some configuration information. This means that each user can create a new project by creating the folder structure and adding his own code. The user then starts the framework with

this project and only sees the code in this project and the sample implementations provided by the framework itself. This limits the size of the framework as users do not add code to the framework implementation.

Additionally, the use of projects makes it easy to exchange simulation results between several users. Two users can simply exchange project folders. The receiver can add it to his local installation of the framework. Like this there are no conflicts between the users, because there is nothing changed in the code of the framework itself.

As the configuration is projects specific and thus provided with each project, each user using a project gets the same results. The results do not depend on the local configuration of the framework but only on the configuration of the project.

Chapter 4

Design and Implementation

In this chapter we present the core design issues of the framework and the concrete implementations of some selected pieces of it.

4.1 Runtime System

As mentioned earlier in this report one of the main aspects of the framework is the performance. We wanted the framework to be as general as possible but, particularly we wanted it to be fast. On the other hand it is sometimes very important to look at the execution of an algorithm to find out if it runs correctly. Therefore, we decided to have a general user interface to control the execution.

To combine the aspects of high performance on one hand and a GUI on the other hand, we decided that the user can switch the GUI on and off. This is not done during runtime but at startup time. The user can either start the simulation with the parameter '-gui' to start it with a GUI or with the parameter '-batch' to have the GUI turned off. See Section A.2.1 for a description of the parameters.

It was very important that the GUI does not influence the performance of the simulation. Thus we decided to completely separate the GUI from the simulation. When the GUI is turned on, the simulation invokes the GUI to refresh itself. The GUI then collects the data from the simulation and displays it. On the other hand when the GUI is not turned on it is not invoked and the simulation can run without any loss of performance. Figure 4.1 shows the separation of the GUI from the simulation.

4.2 Synchronous Simulation

As described in Section 3.1, we decided to just implement a synchronous mode and no asynchronous one. In the synchronous mode, the simulation performs one round after the other. One round means that every node in the system performs one step consisting of the following parts. The system has to update the position of each node, calculate the interference of all messages being sent, update the connections, process the messages, handle the timers and perform the algorithm itself. All this is done for every node in each round. Figure 4.2 shows the lifecycle of one complete round.

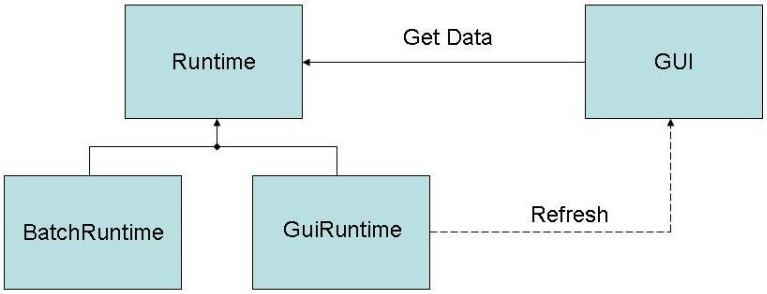

Figure 4.1: The structure of the runtime system. The general user interface is completely separated from the simulation. This ensures that it can be omitted without any loss of performance in the batch mode. The GUI only collects the data when it is refreshed by the runtime system.

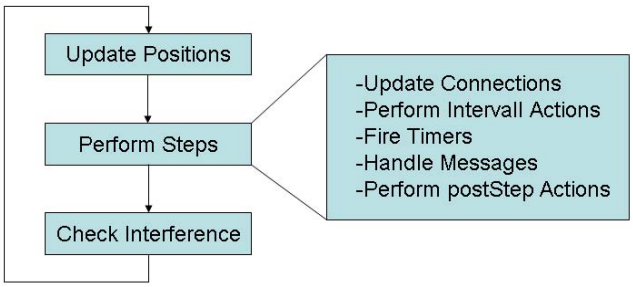

Figure 4.2: On the left side the lifecycle of one complete round is shown. Note that "Update Positions" means that the position of each node is updated and "Perform Steps" means that the framework iterates over all nodes and invokes their step. "Check Interference" means that all the messages currently in the air are checked on disturbance due to interference. On the right side the process of one step on one node is shown. First, the connections to the neighboring nodes are updated according to the connectivity models of the node. Then, the node performs its interval action which means that user specified code to perform in each round is executed. Then, the timers scheduled in this round are fired. Next, the incoming messages are handled according to the code specified by the user. At the end, the postStep actions are performed. This means that user defined code specified to be executed at the end of each step is executed.

Because the execution is not done concurrently but one node is handled after the other, there occur several problems. One, already described in Section 3.2, is that nodes do not know which of their neighbors already performed their step. Another problem occurs due to mobility.

We decided to handle mobility in a separate traverse. This means that we update the position of all the nodes before performing the steps. We handle mobility like this because otherwise we would get problems with consistency.

We don't update the nodes position in the step because this would lead to inconsistent states while executing a round and iterating over all nodes. For instance, if some node already have performed their step of the current round, they are already located at the new position, whereas the remaining nodes still are at the old position. If the connectivity is based on the positions, updating the connections would become a problem, as not all nodes have the updated position information. Figure 4.3 shows an example of such an inconsistent view. Therefore, we ensure that all nodes have accurate position information by calculating it before the steps are performed.

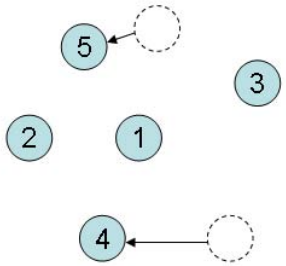

Figure 4.3: An example where an inconsistent view would lead to problems when updating the connections. When node 1 tries to calculate its connections, it has to keep track of whether the neighboring nodes already performed its mobility step in this round or not. In this example the nodes 2 and 3 have not yet performed their mobility step whereas the nodes 4 and 5 already have. Therefore the nodes 2 and 3 still are at the old position and 5 and 6 are already at the new one.

The drawback of our solution is that it decreases the performance because the nodes have to be traversed twice. Once to update the position and once to perform the steps. However, if there is no mobility, the complete traversal can be omitted. Note that there is a configuration entry to completely turn off mobility. So if you switch off mobility, the mobility traversal is omitted and thus is a performance gain.

A very similar problem arises when interference is used. We can not decide in the middle of a round whether a message is disturbed by interference or not. Interference arises when other nodes are sending and thus are generating noise. But the simulation does not know in the middle of the round which nodes are sending in this round. It only knows it from the nodes that already performed their step but not from the others. Thus we decided to keep track of all sending nodes and are handling the interference separately. At the end of each round the system traverses all packets currently being sent and decides for all of them

whether they experience interference or not.

4.3 Interfaces of the Models

In Section 3.3 we presented the purpose of all the models. In this section we present the concrete specifications. It is very important to know them when implementing new models and for understanding the whole simulation. Note that the ConnectivityModels, the InterferenceModels, the MobilityModels and the ReliabilityModels are node specific. This means that there is a separate instance of all these four models saved on every node. The MessageTranmissionModel on the other hand is specified globally and thus there is only one instance of this model for the whole simulation[1]. These are the specifications of the Interfaces of the models.

- ConnectivityModel: The following method is the only method provided by the interface of the connectivity model.

```
public boolean updateConnections(Node n);
```

  The connectivity model is used to update the connections of a node. It has to set the edges to which the node is connected. If there is a connection, it is added to the outgoing connections of the node[2]. This method returns a boolean that indicates whether the connectivity of this node has changed in this round. This means that it returns true when there are new connections or connections disappeared.

  In most implementations the updateConnections method traverses all the possible neighbors and adds or removes connections to the local datastrucuture of outgoing connections. As this traversal is always the same, we decided to add another method to the model to make it easier to implement a new model. When implementing a connectivity model the user doesn't have to implement the updateConnections method but he can overwrite the simpler method

```
isConnected(Node oneNode, Node otherNode);
```

  This method just detects whether two nodes are connected or not. It is called by the system for the actual node and all possible neighbors.

- InterferenceModels: The interference model interface consists of the following method.

```
public boolean isDisturbed(Packet p);
```

[1] The MessageTransmissionModel is saved in the CustomGlobal class which is part of the project specific configuration. See Section 4.5 for details about the CustomGlobal class.

[2] Note that all the connections in the system are directed. This means that the updateConnections method only updates the outgoing connections and not the incomings. Note also that the connections are always calculated according to the connectivity model of the source node of the edge. This means that the outgoing connections are based on the nodes ConnectivityModel whereas the incomings are based on the other nodes ConnectivityModel, which may be different.

This method returns whether the Packet is disturbed by interference or not. It gets called after each round on every message currently being sent. To detect the noise generated by the other nodes, there is a central data structure where all messages currently being sent are stored.

- MobilityModels: The mobility model interface contains the following method.

```
public Position getNextPos(Node n);
```

This method returns the next position of a node. It is called in the mobility traversal on each node in each round. The model can either pre-calculate the route of the node and then return the actual position or calculate the actual position just in time[3].

- ReliabilityModels: The interface of the reliability model contains the following method.

```
public boolean reachesDestination(Node startNode,
                                  Node endNode, Packet p);
```

This method returns whether a packet sent from node startNode reaches the destination endNode[4]. For example, these models can be used for implementing a drop-rate for the connections or other environmental influences.

- MessageTransmissionModel: The interface of the message transmission model consists of the following method.

```
public double timeToReach(Node startNode,
                          Node endNode);
```

This method calculates how long it takes a message to travel from the startNode to the endNode. Normally this will be a constant time but it can also be dependent of the origin and the destination of the packet.

- DistributionModels: The distribution model interface contains the following method.

```
public Position getOnePosition();
```

This method returns the position of one node. The positions all can be pre-calculated. This means that this model can be interpreted as a form of iterator over a pre-calculated distribution of nodes. On the other hand the positions can all be calculated just in time.

[3]Note that the getNextPosition meets the special requirements for the synchronous simulation. In Section 3.1 we suggest to turn off mobility when simulating in asynchronous mode. Nevertheless this method would also meet the requirements for the asynchronous mode as the mobility model can be a function of time and return the positions dependent of the current time.

[4]Although the source node and the destination node are contained in the header of the packet we decided to have the startNode and the endNode as parameters in the interface. It helps understanding the function of this method.

Note that all the interfaces but that of the message transmission model extend the interface Model. This model specifies only one method.

```
public void setParamString(String params);
```

This method sets the parameter string for the model. When creating a model a parameter string can be passed to it[5]. This parameter string is set on the model after a concrete instance has been instantiated. The model can then behave according to this parameter.

4.4 SmartNodeCollection

In Section 3.4 the concept and the purpose of the central data structure were presented. The two important properties for this data structure are an efficient traversing of the nodes and the possibility to get a subset of all nodes which are possible neighbors of a given node.

The main purpose of this simulation is to simulate algorithms on ad-hoc networks like sensor networks or wireless networks whose connectivity models are distance dependent. This means that close nodes are connected whereas nodes at big distance are not. In other words, there is always a radius around a node that divides all nodes in possible neighbors and nodes with certainly no connection to the node in focus. The maximum possible distance two connected nodes may have is hence called rMax. In our implementation of the NodeCollection called SmartNodeCollection we use this fact. We store the nodes in a regular grid. The grid of cell-size rMax is laid over the deployment field and the nodes are stored in the grid according to their position. This enables us to detect depending of their positions in the grid, whether two nodes may have a connection or not. Figure 4.4 shows the organization of the nodes in the grid.

The search for possible neighbors can be done in a very efficient way if the size of the squares of the grid is rMax. This means that possible neighbors of a node may only be in its own square or in one adjacent to it. If we are looking for the possible neighbors we only have to look in these nine squares for possible neighbors. This technique grants the needed properties but also has some drawbacks.

One problem occurs in conjunction with mobility. When a node changes its position, it is possible that it also has to move to another square. This means that after every movement of a node we have to check if it is still in the proper square. This slows down the simulation.

Another and even more constrictive drawback is that the storage is dependent of rMax. This has two consequences. One occurs if someone wanted to change rMax while a simulation is running. In that case the whole data structure would require reconstruction. Especially when there are a lot of nodes in the simulation this would require a lot of time. We decided not to allow a dynamic rMax. This means that rMax is specified on startup and cannot be changed while the simulation is running. If a user wants to change rMax he has to restart the simulation. The other disadvantage is the use of this rMax itself. The user has to specify the maximum range of the nodes on startup. This is a

[5]The parameters are passed either on the command line or by the GUI. See Section A.2.1 for a description of how to pass parameters to models on the command line.

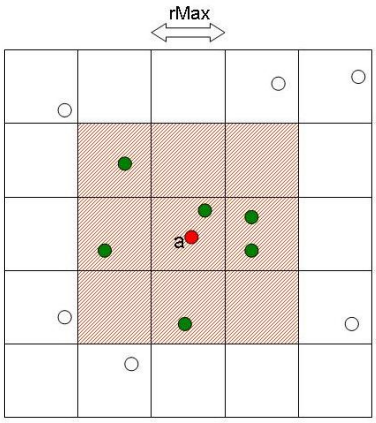

Figure 4.4: We organized the nodes in a regular grid to find out the possible neighbors of a node in an efficient way. The nodes are placed in the grid according to their position on the field. The red node (a) is the node we want to find the possible neighbors for. The side length of the squares is rMax. As rMax is the maximum distance two nodes can have that a connection between them is possible, only nodes in the nine red shaded squares are possible neighbors for the red node. These nodes are the green ones. Note that not all of them have a connection to the red node but for these nodes a connection to the red one is possible. The white nodes certainly have a distance bigger than rMax to the red node and thus aren't possible neighbors of it.

restriction on the connectivity model and a misconfiguration could even lead to undesired simulation results.

On the other hand, the use of rMax helps to restrict the number of possible neighbors the simulation has to check. This increases the performance significantly. We decided that the drawbacks from this solution are less severe than its advantages. Furthermore, if the user does not like the use of this rMax he can set it to a very high value even bigger than the field itself. Then there are no more restrictions to the simulation but the speed performance will drastically suffer from this change.

For the purpose of simulating position based networks this data structure provides the desired properties. Of course, there may also be the need to simulate other networks with different connectivity models not depending of the distance but of other node properties[6]. If this connectivity model makes the SmartNodeCollection unusable, the user can implement his own data structure by implementing the NodesCollection interface and replace the SmartNodeCollection with it. This data structure can depend on any desired property.

4.5 Configuration

One important point of the framework was the usability of the configuration. The simulation results do not only depend of the implementation of the algorithm but they also highly depend of the configuration. For the ease of use it thus was necessary to have the whole configuration collected in one file to have centralized control of the configuration. This file is called ConfigurationXML.xml and is located in the base folder of each project. It is an XML file with two sections. One section is the required section and the other is the custom section. The required section contains the required fields and the custom fields the custom ones. The entries of this xml-file are always formatted in the following way.

```
<Name value=ValueOfField/>
```

The required fields are the configuration properties required by the framework itself. Examples of such fields are the dimensions of the field, the log file name, the initial window size and lots more. On startup these fields are parsed and stored in the configuration class in fields named the same as the xml-tags.

The custom fields are stored in a hash map in the configuration class. They are stored with the name of the entry as the key and the value of it as its value. A user can add custom parameters for his implementations in the custom section of the configurationXML-file and access them based on the tag name. Note that the tags in the configurationXML-file can be nested. An example of a nested entry is shown in the following.

[6]The connectivity could be based on any property of the nodes. For example it could be that there is only a connection between two nodes if they trust each other. For example an implementation of a web of trust.

```
<Custom>
        <SmartNodeCollection>
                <rMax value="80"/>
        </SmartNodeCollection>
</Custom>
```

This example shows the specification of the rMax parameter for the SmartNodeCollection. As it is specified in the custom section the entry gets put in the hash map. It then can be accessed by calling the following method in the configuration class.

```
getIntegerParameter(String key)
```

The entry is identified by the sequence of all the nested entries from the 'Custom' entry separated by a "/". In this example the value of rMax can be retrieved by calling getIntegerParameter("SmartNodeCollection/rMax"').

This mechanism helps centralizing the configurations in one file. Every custom implementation has its parameters specified in this central file. Like this, a user can save the configuration of an interesting simulation by just saving this file. It can even be exchanged between users so that they can reproduce the same simulation results.

For the required fields, there are already default values specified in the configuration class. If there is no entry in the configurationXML file with the same name as the field, these default values are used in the simulation. Obviously the custom fields do not have default values specified. If the simulation tries to access custom field not specified in the xml-file, and thus also not in the hash map, an exception is thrown.

Both the custom fields and the required fields can be overwritten from the console. So if one wants to run the simulation several times with different parameters there is no need to have more than one xml-file. A script can be written to execute the simulation several times with different parameters. See Section A.2.1 for the exact specification of how to overwrite configuration settings from the console.

There is also another file that helps configuring the simulation. It is called `CustomGlobal` and it is also saved in the base folder of each project. It is not directly responsible for configuration of the simulation but it holds project specific global information. This file is a Java class and it holds the termination criterion and the custom GUI functions. The termination criterion basically is a method that is called after every round to find out whether the algorithm has terminated or not. The custom GUI functions are shown in the GUI in the menu called 'Global'. They can be used to interact with the system like for example injecting packets to the system and allows an easy way to extend the GUI.

4.6 Additional Tools

Additionally to the main simulation we provide a handful of useful tools to make the usage more comfortable. These tools are mainly used to collect and log the data and to generate random values. All the classes for the tools in this section are stored in the 'tools'-package.

4.6.1 Data Series

The DataSeries-class provides the user the possibility to collect the data produced by the simulation and to get statistical information about it. Samples can be added to the data series and the mean, the variance or the standard derivation can be computed easily and effective.

4.6.2 Logging

The logging tool is a mechanism to make it easier for the user log information about the algorithm. When outputting with System.out, it is often hard to keep track of all its calls. The key benefit of this tool is that one can specify outputting levels. All the printing statements are then tagged with one of these levels. Before starting the simulation the user can choose which levels should print this time and which don't. This makes the handling of the output very comfortable as the user doesn't need to change all the print statements but can just adjust the outputting level.

The levels are stored in an enumeration called LogL. Each user can add custom levels to this file[7]. There is a threshold level in the class which decides which levels are printed and which aren't. To activate or deactivate a logging level the user then moves the level above or beneath the threshold level.

Depending on a flag called outputToConsole in the configuration the logging is done to the specified log-File or to the console. It is even possible to have more than one logfile to have the information distributed over several files. For example one could log the output of the framework to one file and the output of the algorithm to another file.

Note that this mechanism is somehow c-style. In c/c++ it would be possible to statically check for the logging levels at compile-time and remove unnecessary code. In java there is no such mechanism which means that also the not-printing logging calls are checked, but won't print.

4.6.3 Distributions

A lot of algorithms for node behaviors or models depend on random values. It is sometimes important to be able to reproduce a simulation the user cannot just use the random generator provided by the JVM. We thus decided to provide a whole package for generating random numbers. It does not only consist of a uniform distributed random generator but also of other distributions. There are also a Poisson distribution, a Gaussian distribution and an exponential distribution. This enables the user to get all kinds of random numbers.

The key feature of this packet is that the random numbers generated, of whatever distribution they are, can be made dependent on an initial seed. This seed can be set by the configuration and thus the sequence of numbers generated can be forced to be the same in several simulations. This is particularly useful when debugging a model or a node implementation. When an exceptional behavior occurs the simulation can be restarted identically to the last run and stopped just before the problem occurs.

[7] Obviously, it would be nicer if the LogL class would also be stored in the project and not in the framework itself. But then the logging levels had to be assigned dynamically on startup time. This would make the development tedious because the syntax highlighting and the code-completion are of course based on static information.

Chapter 5

Results

In this chapter we present the results of the thesis. First, we are showing the results of the GUI mode and afterward of the batch mode. Finally we present some performance measurements.

5.1 GUI Mode

First of all, we will present some screenshots of the framework in GUI mode. Figure 5.1 shows the GUI in action. The whole GUI is shown with its two components: the graph panel and the control panel. The graph panel is the left part of the screen displaying the graph.

The control panel is on the right side containing the control elements for the simulation. In the following the control elements are shortly described. On top in a field called 'Round Counter', the number of rounds already performed is displayed. It is not an integer because it is prepared for asynchronous mode where this field would hold the actual time. Below the round information there are the control items. This section is used to control the simulation. The user can enter the number of rounds he wants to perform in the upper field and the refresh rate in the lower one. When hitting the start button the simulation is performing this number of rounds. Note that the abort button only gets enabled when the simulation is running. On the bottom of the control panel the current cursor coordinates are displayed and below that there is the exit button to exit the simulation.

On the left side there is the graph panel. It is a zoomable and scrollable field displaying the graph. The orange rulers on top and on the left side are showing the dimension of the field which helps keeping the orientation on the field. This is especially useful when the user is zooming and scrolling.

Additionally to these two panels there are also two menus. The Graph menu and the Global menu. The Graph menu contains menu entries about the graph. These entries are for Loading and saving graphs, clearing all nodes, generating nodes, getting an info panel and a panel with graph preferences. The info panel is of special interest as it contains information about the current graph. This means that the number of nodes in the graph is displayed and the number and the type of the edges are shown. The Global menu contains the global methods contained in the custom global of the project in use. See Section A.3.3 for

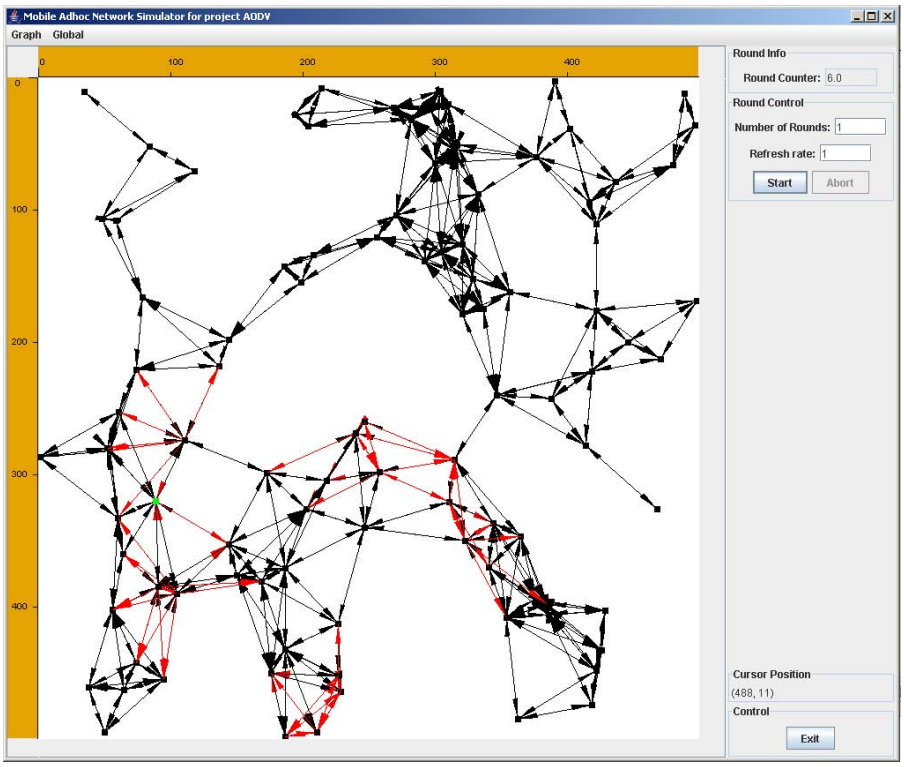

Figure 5.1: A screenshot of the simulation in GUI mode. It is an example of a graph with 100 nodes randomly distributed on a 500x500 field. The connectivity model is UDG with a radius of 80. The red edges are edges on which a message is sent in this round.

information about how to implement custom global functions. Additionally, the global menu also contains the seed used in this simulation.

Figure 5.2 shows another example of the GUI but now with 10'000 nodes on a 2500x2500 field. Displaying 100'000 nodes or even more doesn't make sense as then the screen gets overfilled and the nodes cannot be separated anymore. Use the batch mode to simulate more than 100'000 nodes.

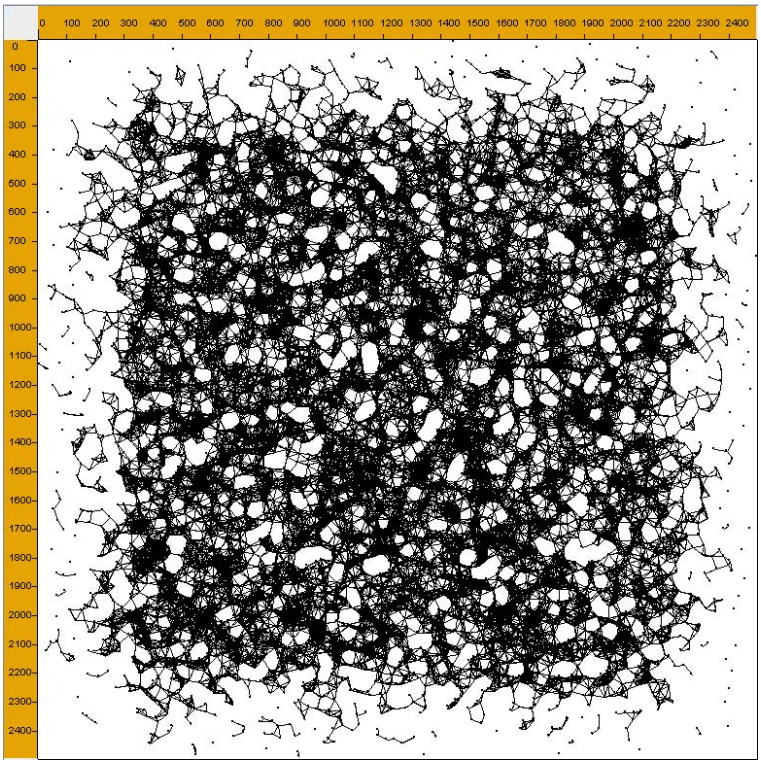

Figure 5.2: This is an example of a graph with 10'000 nodes moving around according to the RandomWayPoint mobility model. The size of the field is 2500x2500 and the connectivity model is UDG with a radius of 45. This screenshot is taken after 150 rounds. Initially the nodes were distributed randomly but note how the density in the middle is now much higher than in the outer regions. Note that we did not draw the arrows in order to increase performance.

5.2 Batch Mode

In batch mode the only output is the logging output. Figure 5.3 shows a screenshot from the output produced by a simulation in batch mode. Note that this is just a sample output. Other simulations can produce different outputs. In this sample we were interested to have some statistical information about the algorithm that was simulated. Therefore we logged the fail-ratios of the diverse messages.

```
        3 reached the target.
        0 died because the flooding failed.
        1 died because the echo failed.
        0 died because the message died.
        1 are still on the way.

Time: it is Mon Jun 12 15:14:51 CEST 2006
Time for this step: 0 ms

Step 119.0 finished.
In this round 44 packets are sent.
Overall 2695 packets are sent.

The mean number of Messages per Step is: 22.647058823529413
This makes 539.0 messages per AODVMessage
Until now the algorithm tried to send 5 Packets
        3 reached the target.
        0 died because the flooding failed.
        1 died because the echo failed.
        0 died because the message died.
        1 are still on the way.

Time: it is Mon Jun 12 15:14:51 CEST 2006
Time for this step: 16 ms

Step 120.0 finished.
In this round 70 packets are sent.
Overall 2765 packets are sent.

The mean number of Messages per Step is: 23.041666666666668
This makes 553.0 messages per AODVMessage
Until now the algorithm tried to send 5 Packets
        3 reached the target.
        0 died because the flooding failed.
        1 died because the echo failed.
        0 died because the message died.
        1 are still on the way.

Time: it is Mon Jun 12 15:14:51 CEST 2006
Time for this step: 0 ms

Simulation regularly stopped after 120.0 rounds during 313 ms
```

Figure 5.3: A possible output of a simulation in batch mode. It was produced by a simulation with 100 nodes randomly distributed over a 500x500 field with dynamic links. The simulation terminated regularly after the specified 120 rounds. The algorithm executed is AODV. The algorithm is specified as follows. When a node wants to send a message to a target node, it floods to discover the route. When it is getting back an echo message from the target it sends the payload message on the discovered route. The output shows the ratio of message transmissions that failed because a flooding message, an echo message or the message itself failed.

5.3 Performance

Because the performance heavily depends on the number of nodes, the specific configuration, the models used, the implementation of the nodes and lots more, it is hard to measure the performance of our framework. To give a rough estimation of the possibilities of the framework, we executed several basic implementations with several numbers of nodes. All the measurements are done on a machine with two Intel Pentium 4 Processors with 3.2 GHz and with 1 GB of RAM. The implementations used to produce the results are shortly presented in the following. As node implementation the DummyNode was used. It is an implementation of a node that does not send any messages. The connectivity models used are UDG and StaticUDG. UDG recalculates all the connections in every round again whereas StaticUDG just calculates the UDG model in the first round and keeps the links static afterward.

Table 5.1 shows the results of some of these measurements with the GUI turned on and in table 5.2 the same measurements for the batch mode are shown.

# Nodes	Connectivity	Node Type	# Edges	Time/Round
1000	UDG	DummyNode	10'000	78 msec
10'000	UDG	DummyNode	100'000	690 msec
100'000	UDG	DummyNode	1'100'000	7900 msec
1000	StaticUDG	DummyNode	10'000	50 msec
10'000	StaticUDG	DummyNode	100'000	220 msec
100'000	StaticUDG	DummyNode	1'100'000	2200 msec
500'000	StaticUDG	DummyNode	4'300'000	8300 msec

Table 5.1: The performance of several simulations with the GUI enabled. The table shows the number of nodes, the connectivity model, the node type, the number of edges and the average time it took to simulate one round. Note that during these measurements the GUI is redrawn every round and that mobility is turned off.

# Nodes	Connectivity	Node Type	# Edges	Time/Round
1000	UDG	DummyNode	10'000	45 msec
10'000	UDG	DummyNode	100'000	490 msec
100'000	UDG	DummyNode	1'100'000	5300 msec
1000	StaticUDG	DummyNode	10'000	4 msec
10'000	StaticUDG	DummyNode	100'000	17 msec
100'000	StaticUDG	DummyNode	1'100'000	78 msec
500'000	StaticUDG	DummyNode	4'300'000	490 msec

Table 5.2: The performance of several simulations in the batch mode. The table shows the number of nodes, the connectivity model, the node type, the number of edges and the average time it took to simulate one round. Note that even with 500'000 nodes and more than 4 million edges the performance is below halve a second per round. Remember that during these measurements mobility was turned off.

For our measurements, we chose graphs of about the same density. Note that the number of edges shows the number of unidirectional edges. This means that in these graphs all the nodes have a mean number of outgoing edges of about 10 and the same number of incoming edges.

Note that in both these measurements the mobility was turned off. Interference was turned on but no interference model was specified. We performed the measurements without mobility to not influence the measurements with the implementation of the mobility model.

The performance of the GUI can be calculated from the two tables above. By comparing the two adequate measurements in GUI and in batch mode the time to redraw the GUI can be calculated. Note how the differences between the corresponding measurements increase with an increasing number of nodes and edges. The GUI needs about 30 milliseconds to draw 1000 nodes with 10'000 edges. For drawing 10'000 edges with 100'000 edges it needs 200 milliseconds and for 100'000 nodes with more than one million edges it needs about 2 seconds. For 500'000 nodes it even needs almost 8 seconds.

Table 5.3 shows the performance measurements for the same graphs as above also with the DummyNode as node implementation but now with mobility. The specified mobility model is the RandomWayPoint model.

# Nodes	Connectivity	Mobility	# Edges	Time/Round
1000	UDG	RandomWayPoint	10'000	55 msec
10'000	UDG	RandomWayPoint	100'000	560 msec
100'000	UDG	RandomWayPoint	1'100'000	6400 msec
1000	StaticUDG	RandomWayPoint	10'000	5 msec
10'000	StaticUDG	RandomWayPoint	100'000	20 msec
100'000	StaticUDG	RandomWayPoint	1'100'000	190 msec
500'000	StaticUDG	RandomWayPoint	4'300'000	1000 msec

Table 5.3: The performance of several simulations in batch mode with mobility. The table shows the number of nodes, the connectivity model, the mobility model, the number of edges and the average time it took to simulate one round.

It is obvious that the overhead for the mobility is dependent of the number of nodes as the framework executes the mobility in each round on each node. Note that it is possible to simulate the 500'000 nodes with over 4 million edges in 1 second per round.

Chapter 6

Outlook

In this chapter we will present the open problems that could not be solved until the end of the thesis. After that we will describe how the framework could be extended. We will also include some important points you need to keep in mind when doing one of these extensions.

6.1 Open Problems

There is basically just one open problem known. It is about the memory management.

6.1.1 Garbage Collection

When a node sends a message, a packet object is allocated. This packet object contains the message itself and a header object. In the header object information about the packet like the sender, the destination, the transmission time and lots more are saved. With high scaled networks of 100'000s of nodes it can easily happen that in every round there are 10'000s of packets exchanged. This means that in every round 10'000 instances of the packet class are instantiated. After having reached the target, the packet objects are not used anymore and the memory used by the packets can be freed. In Java the memory gets freed by the garbage collector. The purpose of the garbage collector is to find out which objects are not used anymore and can be deleted. This means that within a few seconds a high number of packets objects are allocated and deleted.

In tests where we run the simulation over night with a high scaled network with a high network load, the memory was not freed correctly by the garbage collector. We recognized that the memory usage increased over the time. As the memory usage goes back again when lowering the network load the problem cannot be a memory leak.

We noticed that the garbage collector doesn't collect all the garbage packets when the virtual machine still has memory left to use. Obviously this makes sense for normal usages where the user doesn't want the execution be slowed by the garbage collector when there is still memory left. But in our case this generates serious problems. All works fine until the VM reaches the limit of the assigned memory. The simulation then gets slowed down so seriously that

it is not useful anymore. The whole performance is spent on garbage collection and not for the simulation anymore. Sometimes the simulation even runs out of memory because the garbage collector cannot keep up with the pace of the memory allocation.

We tried to solve the problem by manually invoke the garbage collector after every round. This did not solve the problem. Though the memory consumption did not increase as fast as without, it also increases and after letting the simulation run for a long time, the same problems occur.

Apparently, the garbage collector has problems when such a huge amount of short living objects is allocated and deleted repeatedly.

Our proposal to solve this problem is to implement a pool of packets. If the system sends a message, it doesn't allocate a packet but takes an unused one out of the packet pool. When the packet reached its destination it is not needed anymore and can be put back in the pool. Like this the memory usage is not decreased but new packets are only allocated when the pool is empty. As the packets are not deleted but are just put back in the pool the garbage collector doesn't need to collect them.

Concretely this system could be implemented by making the constructor of the packet class private and add a method

```
public static Packet getPacket(Message msg);
```

This method then just checks in the pool for any free packets and returns one of. Instead of clearing the packet buffer after the messages have reached their target, they are put back in the pool of free unused packets.

It is a non-trivial task to implement this mechanism efficient enough that it can handle such a high load.

Keep track of one important problem when implementing such a mechanism. The nodes are not allowed to keep references on the packets after they received them. With this mechanism, the packets are later reused and thus the fields of the packets are changed. This means that, the reference does not contain the same packet anymore.

You could solve this problem by just remember the user not to keep references on packets or you could change the whole message receiving mechanism in such a way that the user doesn't get any packets. This means that the user then just gets the messages and some additional information like for example the sender and the intensity as parameters.

6.2 Future Work

Due to the limited time of this thesis we had to omit some parts of the work. In the following we will present them.

6.2.1 Asynchronous Mode

The most important part of the framework we had to omit is the asynchronous simulation mode. At the beginning of this thesis we wanted to implement a framework being able to work in synchronous and in asynchronous mode. But

we then recognized that it would be a greater effort to also implement the asynchronous mode as we initially supposed. These two simulation modes turned out to differ more than we thought.

The whole simulation process has to be changed, as in the asynchronous mode the nodes are not traversed one after the other but an event queue is handled. The system always takes the next event and executes it. This difference will have effect on other parts of the system. In the following we will present some important points of the asynchronous mode.

We tried to specify the interfaces of the models in such a way that they could also be used in the asynchronous mode. But nevertheless we had to optimize them for the synchronous mode. So it is possible that when implementing the asynchronous mode one has to do some changes on the interfaces. As the specification for the interfaces highly depend on the implementation of the system, it was very hard to anticipate a proper specification of the interfaces. For example specifying the connectivity models for the asynchronous mode is a non-trivial task. Our implementation highly depends of the synchronous mode and specifies whether two nodes are connected in a given round. To generate the connectivity breakup events in the asynchronous mode, the model also has to determine how long a connection will exist. This will most probably cause a change in the interface.

Although we did not implement the asynchronous mode, we had some ideas about the implementation. When implementing interference it would be a great idea to check for interference whenever a node is sending a message. The system then has to evaluate for all messages currently being sent whether some of them are disturbed by the new message. When a message is disturbed the receive event of this message has to be removed from the queue. When mobility is turned off there should be no change in the interference between two sending events.

Another idea came up in connection with how to check for the edges to be present during the whole delivery process of a message. We think that it would be clever to check on connection breakup events if a message gets lost due to this breakup. This means that when a connection breaks up all messages currently being sent over this connection can be removed from the system because their transmission was interrupted. Perhaps this technique could also be adopted for the synchronous mode to get rid of the check in every round.

6.2.2 XML-Graph-file

One smaller point came up at the end of the thesis. The graph-files are now just saved as a plain text file. As graph files are highly structured it would have been elegant to save it as a XML file. Like this it would be much more readable and also better importable by other applications.

6.2.3 Sleep

A function in the API of the nodes is called sleep. We wanted to implement that it is possible to set a node sleeping for a while. This is a realistic behavior as lots of algorithms for sensor networks are developed trying to minimize the power consumption of the nodes. This is often done by setting the nodes sleeping for a while. But this function turned out to be not as relevant as we thought because

this behavior can also be implemented with a timer. Nevertheless the function is still there and one could implement it. Doing this, it is important to specify exactly what sleep mode means. It has to be specified what a node in sleep mode does and what not. For example, it has to be specified if a sleeping node moves and if timer events are handled.

6.2.4 3D

We wanted the simulation to be prepared for tree dimensions. Although not all parts are implemented now, the simulation is prepared for it more or less. The position of a node for example has a third coordinate now always left zero. At some points we had to reduce the calculations to two dimensions for performance reasons. For example when calculating the distance between two nodes the third coordinate is omitted as it would only reduce the performance. When implementing the third dimension it is very important to think about the gui. It is non-trivial to implement a gui for a three dimensional graph.

6.2.5 Logging

The logging mechanism offers the user a powerful tool to easily control the output produced by the simulation. As described in Section 4.6.2 there is a flag to redirect the output either on the console or in a specified file. This mechanism is useful when using just one logger linked to one file. But when more than one logger is used this flag is not that useful anymore. This flag decides for every logger whether to output on console or in a file. The user should be able to take this decision for every logger. What we have in mind is that the user could for example let the system logging appear on the console whereas the algorithm specific logging is written to a file. This is not possible now.

Chapter 7

Conclusion

7.1 The Results

I think that the initial requirements are well met. We created a framework in which algorithms can be executed and tested in a very dynamic environment. We managed to specify the interfaces in a very simple way. This helps the user understand the system and allows a fast development. Nevertheless, the simulation is fast and even simulation on high scaled and very dynamic networks with high network load perform very fast.

Although the initial requirements were high, we managed to achieve almost all of them. The only task we could not finish was the asynchronous simulation mode.

7.2 Personal Experience

The work on this thesis was very interesting but also very challenging.

It was the first time I implemented such a big project. Finding out the requirements for the framework, specifying the interfaces and discussing about the use cases was very interesting. It certainly was a very instructive experience to write such a big project from the beginning. I learned a lot about how important it is to have a proper design from the beginning and how important it is to work very precisely.

Additionally, I could improve my knowledge about the Java programing language. It was very absorbing to get to know the new features of the Java 5.0 version. Also the use of reflection was new to me.

Appendix A

User Manual

In this chapter we will present a short user manual. We tried to cover all the essential points. If you don't find information about a particular part of the code here, please check the code for comments. In this chapter we will first present instructions about the installation. After that some information about the general usage of the framework is presented. In the following , we show how the development of new models and algorithms works and to what one should turn his attention. Finally we are presenting some tips about the usage of the framework.

A.1 Installation

The whole project was written in Java using the Eclipse[1] IDE. There are Eclipse-specific configurations so if you are using Eclipse too, you can just open the ManS-Project and don't have to change the configuration. Alternatively you could compile the source with a script called compile. It automatically creates the folder structure and compiles with the correct options for the VM[2].

If you are using another IDE you should properly configure the output path for the binaries. All the class files should be compiled to the folder 'binaries'. Set the output folder for the project classes to 'binaries\projects' and the output folder for the sources of the framework to 'binaries\source'.

Obviously an installed Java-VM is needed to run the framework. We developed the framework with the Virtual Machine from Sun[3]. There are some features used in the framework which are only available since Java 5.0. So install Java 5.0 or later.

[1]Eclipse is a free available open-source Java-IDE. You can download it from www.eclipse.org

[2]Both the jdom.jar for XML parsing and the path to the class files have to be included. Note, that the class files for the framework are located under 'binaries\source' and these of the projects are in the 'binaries\projects' folder.

[3]Download it from java.sun.com.

A.2 First Steps

There are several possibilities to start the framework. First of all you can start it in your IDE by executing the main method in the class Main. Alternatively there are two scripts to start with. For Unix/Linux or Cygwin users there is a script called run. For Windows users there is a batch file called run.bat. Start the framework by executing one of these scripts. Note that the run script takes parameters and passes them to the application. In the run.bat script the parameters have to be specified in the script itself.

A.2.1 Parameters

Independently of the way you start the framework, you have to specify some parameters. The parameters consist of the following options:
options = [-help|-gui|-batch] {-project|-gen|-graph|-rounds|-refreshRate|-overwrite}

The available options are presented in the following.

- -help: When this option is selected, the information about the parameters is shown.
- -gui, -batch: One of them is required to be the first parameter. It indicates whether to start in GUI- or in the batch-mode. GUI mode means, that a GUI is created and the simulation is shown in it. Batch mode means that there is no GUI shown.
- -project projectName: Use a specific Project. The framework is meant to always be used with a project. The system will print out a warning if you do not specify one. Use projects to keep track of your implementations and the actual configuration. Projects are described in detail in Section 3.5 and their creation in Section A.3.1.
- -gen numberOfNodes DistributionModel TypeOfNode ConnectivityModel InterferenceModel MobilityModel ReliabilityModel:

  Generate a graph at startup according to the parameters set. All of them have to be specified. The generated graph then consists of 'numberOfNodes' nodes of type 'TypeOfNode' distributed over the field according to the DistributionModel specified. The other four models specified are used for all the nodes[4]. Note that you can have more than one -gen sections so you can use a graph consisting of several types of nodes. You can pass a parameter to each model by just appending a bracketed string. For example one could select a DistributionModel by writing MyDistributionModel (my parameters). The parameter string is then passed to the model after having instantiated the instance. Note that all the instances of the models get the same parameter string set.
- -graph graphFile: Load a graph file at startup. Note that in the graph file the connections are saved and when you load it, they are also drawn. But if the connections do not fit with the actual configuration they are removed in the first step after loading. This means, that you do not save a fixed graph but more a snapshot of a simulation.

[4]This doesn't mean that there is one global instance of the models for all nodes but that they all use an instance of the same class.

- -rounds numberOfRounds: Perform this number of rounds. In batch mode the simulation performs this number of round and then exits. Note that when running in batch mode you should specify the number of rounds to perform because otherwise the simulation does nothing[5]. In GUI mode it performs this number of rounds and then halts. It does not exit. So you can perform another number of rounds afterward. You can also abort the simulation between two rounds.

- -refreshRate rate: This parameter specifies the rate the GUI updates itself with. This means that the GUI is only redrawn all 'rate' steps. This does not have any consequences on the simulation itself but it makes it faster when the rate is set higher. Especially if you are only interested in the evolution of the simulation and not in every round, this options makes sense. Note that this parameter has no meaning in batch mode.

- -overwrite parameterName=newValue: With this option you can overwrite the value of a variable in the configuration class. The system looks for a member-variable with the specified name in the configuration class and if it finds one, it overwrites the value of it with the specified one. Note that this mechanism also allows to overwrite values set by the configurationXML-file.
  If there is no field with the specified name the value is just puts it to a collection of parameters. See Section 4.5 for details about the configuration and Section A.3.2 for its usage. Note that you can have more than one occurrence of the -overwrite option. You can also specify more than one name=value pair to overwrite behind one -overwrite option. All the pairs until the next known option or the end of the parameter string are processed.

A sample parameter string using the options specified above is shown in the following.

```
-gui -project AODV -gen 10000 Random AODV:AODVNode AODV:DynamicLinks
  InterferenceModel MobilityModel ReliabilityModel -overwrite
  logFileName=projects/AODV/overnight1.txt outputToConsole=false
  -rounds 1000 -refreshRate=10
```

Calling the simulation with this parameter string, it will start in gui mode, use the project called AODV and generate 10000 randomly distributed nodes with the specified models. It overwrites the logfile name and sets redirects the output to the logfile. It performs 1000 rounds with a refresh rate of 10.

[5]When you want to run the simulation until the termination criteria holds, run the simulation with a very big number of rounds.

A.3 Usage

After the first steps with the default implementations you can start implementing algorithms yourself. In the following the steps to implement algorithms are presented.

A.3.1 Creating Projects

The first step when starting to implement algorithms is to create a new project. It is important not to create your own classes in the framework itself. If every user would add custom code to the framework the amount of code for the framework would increase with every new user. Additionally it would be difficult to keep track of what part of the code is really essential for the framework and what has been added by a user. It is also much easier to understand the usage of the framework if only the original classes are present.

You can also create more than one project. In fact it is most useful if you are creating a new project for every algorithm you want to simulate. Like this it is easy to keep track of the configuration for the simulations because they are project specific. Also other properties like the global functions and the termination criteria are stored project specific.

In the following the structure of the projects is described. Figure A.1 shows the folder structure of the sample Project. This is a complete project with all the important folders. Note that you can omit folders in your project if they are empty.

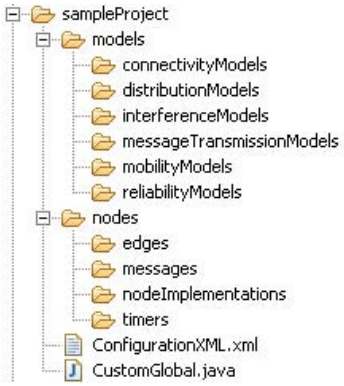

Figure A.1: This is the folder structure of a complete project.

All the folders are named according to their designated content. You should always save your classes in the appropriate folder to allow the system to access them. For example you should store your self-written connectivity models in the folder named models/connectivityModels and the implementations of the nodes in the folder nodes/nodeImplementations. When addressing one of your classes in the system[6] you can always address them by ProjectName:Classname. This means that when ever you are identifying a self-written class you can address them like this. This is mainly done to prevent name conflicts. When

[6]For example when generating nodes on the command line at startup.

accessing a mobility model named UDG, the system searches for the class file in the appropriate folder of the frameworks folder structure. On the other hand when accessing a model named AODV:DynamicLinks the system automatically searches in the appropriate folder of the project named AODV. Note that in the GUI mode when generating or creating nodes, only correctly stored classes are shown. To reduce the number of models displayed in the GUI only models of the currently used project and those of the framework are displayed.

A complete project consists of the folder structure and two files. The ConfigurationXML.xml file is described in Section A.3.2 and the CustomGlobal file in Section A.3.3.

The easiest way to create a project is to copy the sample project and rename the root folder of the project to your project name. Note that you also have to change the packet name in the CustomGlobal class from sampleProject to the name of your project.

When you want to exchange simulations with other users it is most easy to exchange the whole project folder because the whole configuration is then also included and the other user can completely reproduce your simulation. Note that you can also address a class from other projects in the same way as addressing your own classes. The framework then searches for this project and the specified class in it.

A.3.2 Modifying the Configuration

One of the most important points when using the framework is to set up the configuration. There are three different places to specify the values of a configuration. First there are the default values specified in the configuration class. You should not change the values there unless you are sure, that they are completely wrong. The second possibility is to have it specified in the configuration-file (named ConfigurationXML) of your project. Change the values here when all the simulations specified in your project use the same values. For example if you are simulating a routing algorithm for static networks you can turn off mobility here. The third possibility is to specify the values on the command line on startup with the -overwrite option[7]. This possibility is meant to be used if you are having configuration entries used only in one simulation or if they are changing in every simulation. Note that the parameter specification is obviously not passed to other users when exchanging a project.

It is important to know the priority of how these parameter specifications are treated. The default values are overwritten by the specified values form the ConfigurationXML file. These values are also overwritten by the values specified on the command line. So the priority is increasing from default values to ConfigurationXML file and to command line specification.

Summarizing, it can be said that if you want to specify settings for all implementations in the current project you specify them in the xml file. For settings concerning only one run, like the name of the output file, they should be specified by overwriting them from the console. Settings in the XML files are project specific and settings from the console are simulation specific.

Additionally to the configuration file there is another file which you can use to specify the environment. It is called CustomGlobal and is also located in

[7]This option is described in detail in Section A.2.1.

the root folder of your project. In this file you can implement the termination criterion. The function to overwrite is called hasTerminated. It is called after every complete round to find out whether the simulation has terminated or not. There is a method to save properties about the termination. It is called setTerminationProperty and can be called from wherever the information about the termination is coming from. Normally the hasTerminated method would access some static fields on the nodes to find out whether the algorithm has terminated or not.

In the CustomGlobal file you can also specify globally available functions. These functions all have to begin with a predefined prefix. Per default the methods have to begin with 'GLOBAL_METHOD_' to be accepted as a global method. These methods are available in the gui in the Global menu. You could use these methods for example to start the algorithm.

A.3.3 Implementing Algorithms

Basically implementing an algorithm means implementing the behavior of the nodes and specifying the behavior of the environment. Implementing the behavior of the nodes is done by extending the nodes class and implementing its methods. The environment is specified by models. These models specify how the nodes are connected, how the nodes move, how reliable the connections are and how the interference affects the sending process of messages. You can implement a model by extending the superclass of the model and by implementing its methods. Check the documentation of the models to get information about the functionality of the methods.

When implementing custom nodes you have to know the functioning of the simulation. For the exact node behavior check the calling sequence in the handle() function of the node class and the appropriate runtime thread. The node behavior is then implemented by overwriting the following methods declared abstract in the Node class.

```
void handleMessages(Enumeration<Packet> arrivingPackets);
void performIntervallStep();
void init();
void neighborhoodChange();
void postStep();
String toString();
```

The **handleMessages(Enumeration<Packet>arrivingPackets)** method is called in every step and by overwriting this method you specify what the node has to do when there are messages incoming in this round. Note that this method is called with an empty enumeration when there are no arriving messages for a node in this round.

The **performIntervallStep()** method is also called in every step and specifies the regular action that the node takes in every round.

The **init()** method is called at the beginning of the lifecycle of a node. It may be used to initialize the start state of the node. Note that this function may not depend on the neighborhood of the node as the init function is called before the connections are set up.

The **neighborhoodChange()** method is called when there was a change in

the connectivity of a node. This means that a connection adjacent to this node was added or removed.

The **postStep()** method is called at the end of the step of a node. It is mostly used for logging or for cleaning up.

The **toString()** returns a String representing this node. It is used for the information dialog and the popup menu for the nodes in the GUI.

By overwriting these functions you implement the behavior of the node. It is important to implement the functions in an efficient way. Especially the performIntervallStep() function is very performance consuming when lots of nodes are used. Try to minimize the code executed in this method.

When implementing the behavior of the nodes you can use the following methods to send messages.

```
public void sendPacket(Packet p, int t)
public void sendPacket(Packet p, int t, double i)
public void sendMessage(Message m, int t)
public void sendMessage(Message m, int t, double i)
public void sendPacket(Packet p, Node t)
public void sendPacket(Packet p, Node t, double i)
public void sendMessage(Message m, Node t)
public void sendMessage(Message m, Node t, double i)
public void broadcastPacket(Packet p)
public void broadcastPacket(Packet p, double i)
public void broadcastMessage(Message m)
public void broadcastMessage(Message m, double i)
```

The methods are all more or less the same but with different parameters. You can send or broadcast messages or packets. The target node can be specified by its ID or by the node itself. All methods are also available with a intensity parameter. This intensity describes the power of the sending radio module and may be used for interference. Note that sending messages is preferred over sending packets because memory can be saved.

Often when implementing nodes, timers are used. You can implement a timer by extending the timer class. Timers are started by the startRelative or the startAbsolute method. These methods start the timer either in a relative or absolute time. Relative means that the timer is started in the specified amount of time counted from the actual time of the call and absolute means that the timer is started on the specified time. When the time to fire has come, the timer invokes its fire() method. Note that in the synchronous mode timers fire in the next round after the time has expired when a non integer time is specified. So if you set the timer to start on time 5.5 it fires in the 6th round. When the time is set exactly on a round it of fires in that round.

A.3.4 Maps

When using the framework you can also use maps. Maps are images that can be displayed in the background of the simulation. They are defined by a two dimensional array of integers. The dimension of the array is specified on the first line. The mapping of numbers to colors can be made in the map class. The background image is divided in squares according to the specified map file. Each entry in the map file specifies the color for a whole square.

It is not the only purpose of the maps to have a background image, but the algorithms can also react on the values in the map file. You can access the value of a position with the getValueAtPos(Position pos) method in the map class. The method returns the value of the map file that is projected on the specified position. For example you can implement a mobility model for the nodes to route around some sort of obstacles like lakes. Figure A.2 shows how a map is projected on the background of the simulation.

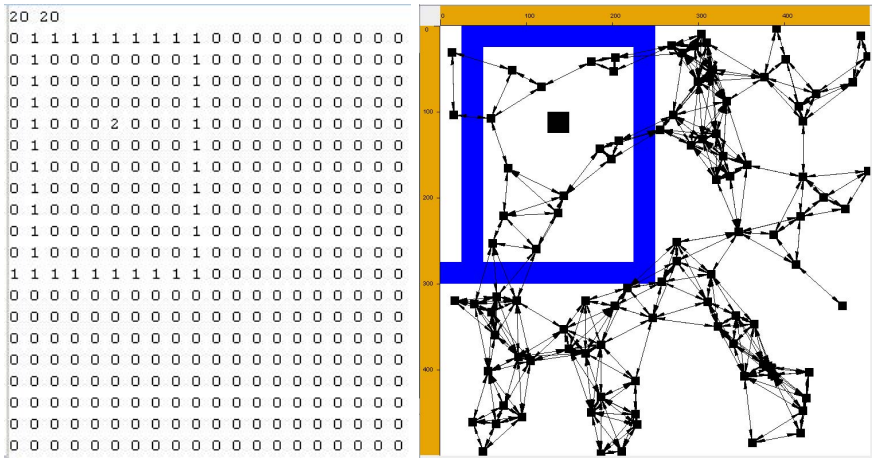

Figure A.2: This is a sample of a map file. The specification of the map file on the left side results in a background image like the one on the right side. Note that in this example 0 means white, 1 means blue and 2 means black.

A.3.5 Sample Application

In the following we will present the process of developing a sample application. We start with a empty node and try to illustrate the possibilities of the nodes. We are implementing a very simple algorithm but try to cover the most important points of the node development. We want to implement nodes that start flooding an IntMessage with their own id with a given probability. The nodes re-broadcast an incoming message if it is the first message the node receives. Otherwise it will do nothing. This will somehow implement a algorithm with which every node gets to know the id of the closest node that initially started the algorithm.

We begin with a empty node class. This means that we are starting with a class FloodingNode that is extending the Node class. The seven methods are the inherited methods declared abstract in the superclass.

This is the class we are starting with:

```
public class FloodingNode extends Node {
 public void handleMessages(Enumeration<Packet>
                                arrivingPackets){}
 public void performIntervallStep(){}
 public void init(){}
 public void neighborhoodChange(){}
 public void postStep(){}
 public String toString(){return null;}
 public void checkRequirements() throws
                                WrongConfigurationException{}
}
```

We want that the nodes start flooding in the third round. This requires the implementation of a timer. We extend the Timer class and implement a timer that, when it fires, broadcasts a IntMessage with the id of the node as value. The implementation of the timer class is shown in the following.

```
public class BroadcastTimer extends Timer {
 public void fire() {
  node.broadcastMessage(new IntMessage(node.ID));
 }
}
```

When the timer fires it executes its fire method and this method starts broadcasting a IntMessage on its node.

In its init() method the node can create a BroadcastTimer and start it in the third round. Therefore the startAbsolute() method is used. The implementation of the init() method is shown in the following.

```
public void init() {
 Timer bcastTimer = new BroadcastTimer();
 try {
  bcastTimer.startAbsolute(3, this);
 }
 catch (MisconfiguredTimeException e) {
  e.printStackTrace();
 }
}
```

As we want the nodes to start flooding with a variable probability, we added a parameter to the custom section of the configurationXML.xml file of our project. This parameter is specified in the following.

```
<Custom>
        <FloodingNode>
                <probability value="5"/>
        </FloodingNode>
</Custom>
```

The sending method can generate a random number generator with the getRandom() method of the Distribution class. The random value generated by this random number generator is compared with the parameter value. The parameter value can be accessed with the getIntegerParameter() method of the Configuration class. The parameter is identified by the names of the two nested tags ¡FloodingNode¿ and ¡probability¿.

The node compares the generated number with the one stored in the parameter and only when the generated number is smaller, the node starts his timer. The modified init() method is shown in the following.

```
public void init() {
 try {
  Random rand = Distribution.getRandom();
  int randomValue = rand.nextInt(100);
  int parameter = Configuration.getIntegerParameter(
                                    "FloodingNode/probability");
  if(randomValue < parameter){
   Timer bcastTimer = new BroadcastTimer();
   bcastTimer.startAbsolute(3, this);
  }
 }
 catch (CorruptConfigurationEntryException e1) {
  e1.printStackTrace();
 }
 catch (MisconfiguredTimeException e) {
  e.printStackTrace();
 }
}
```

Now we have nodes that start broadcasting an IntMessage with a probability specified in the configuration. We now implement the forwarding of the messages. All node have a flag indicating whether the node already received a flooding or not. This flag is called *received*. When *received* is still false the node re-broadcasts the message. The implementation of the handleMessages() method is shown in the following.

```
public void handleMessages(Enumeration<Packet> arrivingPackets) {
 while(arrivingPackets.hasMoreElements()){
  Packet packet = arrivingPackets.nextElement();
  if(!received){
   IntMessage intMsg = ((IntMessage)packet.message);
   id = intMsg.value;
   received = true;
   this.broadcastPacket(packet);

   Global.log.logln(LogL.ALGORITHM_LOGGING, "Node "+this.ID+
        " received the flooding from "+packet.header.origin.ID);
   Global.log.logln(LogL.ALGORITHM_LOGGING,
        "The next flooding node is: "+id);
  }
  else{
   Global.log.logln(LogL.ALGORITHM_LOGGING, "Node "+
    this.ID+" received a flooding from "+packet.header.origin.ID);
   Global.log.logln(LogL.ALGORITHM_LOGGING,
    "But it already has a closest flooding node");
  }
 }
}
```

The method iterates over the incoming messages and checks, whether the received flag is set. When the flag is set false, the message is casted to an IntMessage and its value is stored. It sets the received flag and re-broadcasts the packet. Depending on the received flag the node logs some information about its state.

To get some information about the node in the GUI, the toString() method is also implemented. If it already received a flooding, it returns a String containing

the ID of the closest flooding node. Otherwise it returns en empty string.

```
public String toString() {
 if(received){ return "My closest flooding node is "+this.id; }
 else{ return ""; }
}
```

After having implemented all the methods of our node, we can start the algorithm by passing the following parameters to the simulation.

```
-gui -project Testing -gen 100 Random Testing:FloodingNode UDG
   InterferenceModel MobilityModel ReliabilityModel
```

A.3.6 Tips

In this section we will present you some tips that could be useful when using the framework.

One important point when using the framework is the proper configuration of rMax. To help you with the configuration we will present you some settings that helped us to keep track of the graph used. Table A.1 shows some rMax settings to get a medium dense network. Medium dense means that the mean number of outgoing and incoming edges per node is approximately 10. In all the settings the UDG connectivity model is used. We found out that it is nice to have a table like that because especially in batch mode the configuration is not easy.

# Nodes	Field size	rMax	# Edges
100	500x500	100	1000
1000	2500x2500	150	10'000
10'000	2500x2500	45	100'000
100'000	2500x2500	15	1'000'000
500'000	2500x2500	6	4'500'000

Table A.1: This table shows the rMax to choose to get a medium dense network. The settings are shown for 100, 1000, 10'000, 100'000 and for 500'000 nodes. The connectivity model chosen is the UDG model. Note that all the edges are unidirectional and thus these settings result in nodes with approximately 10 incoming and 10 outgoing edges.

Another tip is to always overwrite the name of the logfile used for a simulation. When specifying the logfile only in the configurationXML file all the simulations using the same project access the same logfile. It just overwrites the logfile and does not check for its existence. When you don't overwrite the name of the logfile it will often happen to you, that you forget to change the value in the xml file and thus you will lose old simulation results. Therefore, always overwrite the logFileName parameter with the -overwrite option. Alternatively you can use another instance of the logger in your implementation that is writing in a file with a timestamp in the name. You can do this by writing the following in your code.

```
Logging.getLogger("output file of "+(new Date()).getTime());
```

Like this you can log to this logger which writes to a file with a timestamp.

A.3.7 Tuning

If you are using the framework and you need more performance, there are several possibilities to tune the simulation.

The first one is to give the JVM more memory to use. Use the JVM option[8] -Xmx. For example you could use -Xmx700m to give it 700 MB instead of the usual 64 MB. This will increase the memory that is available and thus the garbage collection time is decreased. This makes the simulation significantly faster. Note that you should not assign the VM more memory as available. Do not use the virtual memory because when the simulation starts swapping out data of the simulation it gets very slow.

Another tip is about the usage of the packets. As stated before the sendMessages methods are preferred over the sendPackets methods because no additional package is allocated. For security reasons the system clones the packet passed and afterward the passed packet is not used anymore. This increases the garbage collection load. On the other hand most of the calls of these methods are in a concrete implementation of the handleMessages method of a node. Often the received packets are forwarded directly. To avoid generating unused packets just reuse the received packets by passing them to the sendPacket method.

A similar problem occurs with the messages. Whatever method you use to send a message, the system clones the message for security reason. This cloning is done to ensure that no references on the messages exist anymore. This prevents that messages that already have been sent are edited. If you are sure that no one misuses the package references, the clone() method of your message can be implemented by just returning **this**. Like this no additional messages are allocated when sending and memory is saved. Especially when your messages have a big payload size this technique saves a lot of memory.

[8] Check the following url for more information about the parameters for the Java HotSpot VM. http://java.sun.com/docs/hotspot/VMOptions.html

References

[1] Stefan Schmid and Roger Wattenhofer. Algorithmic Models for Sensor Networks. In *14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS), Island of Rhodes, Greece,* April 2006. `distcomp.ethz.ch/publications/wpdrts06cam.pdf`

[2] Nicolas Burri, Roger Wattenhofer, Yves Weber, and Aaron Zollinger. SANS: A Simple Ad hoc Network Simulator. In *World Conference on Educational Multimedia, Hypermedia Telecommunications (ED-MEDIA), Montreal, Canada,* June/July 2005.

`http://dcg.ethz.ch/publications/edmedia05.pdf`

[3] Homepage of the Distributed Computing Group of the Swiss Federal Institute of Technology (ETH) in Zurich.

`www.dcg.ethz.ch`

[4] Homepage of the GloMoSim project.

`pcl.cs.ucla.edu/projects/glomosim/`

[5] Homepage of the Information Science Institute of the University of South California.

`www.isi.edu`

[6] Homepage of the NS2 Network Simulator.

`www.isi.edu/nsnam/ns/`

[7] Homepage of the Parallel Computing Laboratory of the University of Coalifornia in Los Angeles.

`pcl.cs.ucla.edu/`

[8] Homepage of the Parsec Programming Language.

`http://pcl.cs.ucla.edu/projects/parsec/`