Semester Project: Distributed FRANC Simulator

Aurélien Frossard (aurelien.frossard@epfl.ch)
Boris Danev (boris.danev@epfl.ch)

Winter semester 2003

Distributed Systems Laboratory (LSR)

David Cavin (david.cavin@epfl.ch), Supervisor
Yoav Sasson (yoav.sasson@epfl.ch), Advisor

Abstract

Simulation and emulation are one of the powerful tools for evaluating the correctness and testing the performance of wireless Mobile Ad-hoc Networks (MANETs). The need for simulators/emulators comes from the fact that MANETs are highly dependent from the physical environment. This particularity does not facilitate the research community into exploring and improving the MANETs' behavior. This paper presents a simulator developed specifically for FRANC[1], a Java framework for development and evaluation of wireless mobile ad hoc networks.

[1] FRamework for Ad-hoc Network Communication

Contents

List of Figures

1 Introduction

1.1 The project

The Distrubuted FRANC Simulator was a semester project supervised by David Cavin and Yoav Sasson, PhD students at the Distributed Systems Laboratory (LSR) at the School for Computer and Communication Sciences (IC) of the Swiss Federal Institute of Technology in Lausanne (EPFL), under the direction of Professor André Schiper.

The project was realized by Aurélien Frossard and Boris Danev, senior students in the School of Computer Science.

The simulator allows to simulate an *ad-hoc network topology* of nodes based on FRANC[2], a Java based framework for development and evaluation of wireless ad-hoc networks. The framework is currently under development and the idea to enable FRANC with simulation capabilities came naturally during the development in order to be able to test new ideas, architectures, and possible optimizations or simply implement applications. The Distributed FRANC simulator is supposed to respond to the needs just mentioned.

1.2 MANETs and FRANC

MANET stands for Mobile Adhoc NETwork. It is a wireless network where members do not depend on any wired infrastructure to communicate with each other, nor are they required to stay at a predefined position. The topology of such a network is not fixed, it changes over time.

FRANC stands for FRamework for Ad-hoc Network Communication. It was designed by the Distributed Systems Laboratory (LSR) at EPFL. Its purpose is to facilitate the development and evaluation of wireless mobile ad hoc networks.

For more details on FRANC, please refer to two excellent papers on the subject:

[1] *FRANC: A lightweight Java Framework for Wireless Multihop Communication*
[2] *Semester Project: MANET Framework*

[2] FRamework for Ad-hoc Network Communication

1.3 Paper organization and intended audience

The paper is organized in eight chapters which we briefly discuss below:

- **Section 1** is what you have just read. It provides general information about the project, its premises, authors and supervisors. It also briefly discusses FRANC, a Java framework for development of wireless ad-hoc networks.
- **Section 2** is an overview of the project itself and describes different assumptions, approaches and personal objectives.
- **Section 3** is dedicated to the architecture of the simulator.
- **Section 4** goes in depth in the different design and implementation decisions involved in the construction of the simulator's components.
- **Section 5** provides information about the testing strategy used in proving the correct behavior of the simulator.
- **Section 6** describes how to start a simulation process (network configuration, logging, etc). It also provides a simple example of simulation process.
- **Section 7** gives an insight of current limitations and future developments.
- **Section 8** discusses some of the lessons learned during development.
- **Section 9** provides a final conclusion.

The paper is intended for people who want to simulate network behavior based on FRANC, as well as for those who want to get a grasp of the problems and design in building a distributed network simulator.

2 Overview

The project presented two major groups of objectives. The first group was tied to the challenge of designing and building a distributed simulator in Java, with all the vicissitudes of the thread management and real-time behavior of this language. The second group was the purely personal challenge of working in a team, finding the best tools for development and improving our experience through building a complex multi-threaded Java application.

2.1 Project objectives

2.1.1 Assumptions

The whole simulation process is based on the assumption that each simulation node is implemented using the FRANC multi-layer framework.

We decided to implement the simulator using the latest stable Java specification version 1.4.2, in order to benefit from its better performance, security and improved network, reflection and thread management.

We decided to use a distributed architecture in order to allow the simulation process to be deployed in a cluster of machines for best scalability, precision and performance. However, the simulation process could be run on a single machine but the network topology in this case should be of limited size depending on the quality of the machine. On the other hand, if the simulation process is deployed in a distributed environment, we assume good clock synchronization (1 - 10 ms) based for example on NTP[3]. The Distributed System Laboratory has a cluster of 16 machines which fulfill the requirements mentioned above.

We decided not to bind the simulator to any particular technology for distribution of the simulation information during startup. Thus, any user should evaluate its available environment and write the according scripts needed to deploy the simulation.

2.1.2 Approach

In terms of approach of development, we adopted the standard for Java, Object-Oriented development approach which includes the following phases:

Phase 1 Analysis (OOA) includes defining the needs of the application, create

[3]Network Time protocol [11]

possible use cases, identify the actors and the system operations, and elaborate the project scope.

Phase 2 Design (OOD) includes brain-storming to extract the different application modules, their interaction and dependencies.

Phase 3 Programming (OOP) includes all the implementation issues.

Phase 4 Testing Object-Oriented Software (TOOS) includes the testing strategy adopted and proves the correct behavior of the application.

Phase 5 Documentation.

2.1.3 Tasks

The main task for the project was to enhance FRANC with simulation capabilities. Given this general idea, we identified the following list of tasks:

- Brain-storm and define the desired functionalities of the simulator. This task was to be done with the collaboration of David Cavin and Yoav Sasson, who had specific needs depending on their research work. We decided also to look at some already developed ad-hoc simulators/emulators in order to get ideas about interesting and useful functionalities.

- Evaluate and define the type of simulation: real-time or discrete. We needed to look at these two possible simulation techniques, evaluate them according to the programming language limitations and take the best suited one for us.

- Evaluate and define a type of architecture: local or distributed. This task involved testing the number of FRANC nodes that could be run on one machine, and evaluate the advantage of using a distributed architecture.

- Discuss and design a modular architecture with possible extension points in order to give the possibility to customize or completely change important features. Thus, design a kind of simulation framework and implement one possible solution.

- Research and evaluate different possible solutions for logging the information generated by the simulation process. Identify what to log, where and how.

- Define and use a testing strategy which allows individual and composite module testing. Test the correct behavior of the implemented solution.

- Prepare a demo simulation process which shows clearly the functional behavior of the simulator and its performance.

2.2 Personal objectives

The project proposal gave the possibility to work in a group, which allowed us to experience a real software development process. Thus, we had to show an excellent cohesion and good teamwork, and prove that working in a team is a very nice experience for our future work as computer scientists. Given the advantage to work in a team, we wanted to experiment new development techniques, explore and integrate new tools which would enrich our computer science background.

2.2.1 Teamwork

Why being two on a semester project first of all? Well, we simply think it brings more fun to the task, more motivation to both the participants and better quality to the final product.

However we do not think teamwork means *extreme programming* or *code review*. Finding a solution or coming up with ideas is a result of an individual thinking process. Finding the best one comes through confrontation and exchange between people: one can never find alone all the solutions to a problem, or sometimes the flaws of its solution, but others usually can. That is why we always questioned each other whether an idea is good, bad and to what extent. We accomplished most of the work separately and met only when critical tasks, problems or decisions had to be addressed. In other words, we designed most of the software together, but implemented each part alone.

2.2.2 Separation of work

Boris was responsible for the main simulator module, the communication between coordinator and simulation nodes, and the clock/controller architecture, the configuration of the simulation process, and the *Log4j* and *JDom* integration.

Aurélien worked on the communication between nodes, the integration of *NS2's* mobility patterns, the event system and the first version of their logging mechanism. He also contributed on the integration of *Eclipse* as our development tool.

Together with David Cavin and Yoav Sasson, we discussed and approved all important conceptual, design and implementation decisions.

2.2.3 Technologies and tool integration

During the development process, we tried to explore and integrate in the project different tools which purpose was to facilitate our development process. Three of them were of particular importance in terms of efficiency:

- [8]*JDOM* A Java Document Model package with a fully object-oriented view of XML.
- [9]*Log4j* A powerful logging package, currently the de facto standard.
- [10]*Eclipse* Java Integrated Development Environment.

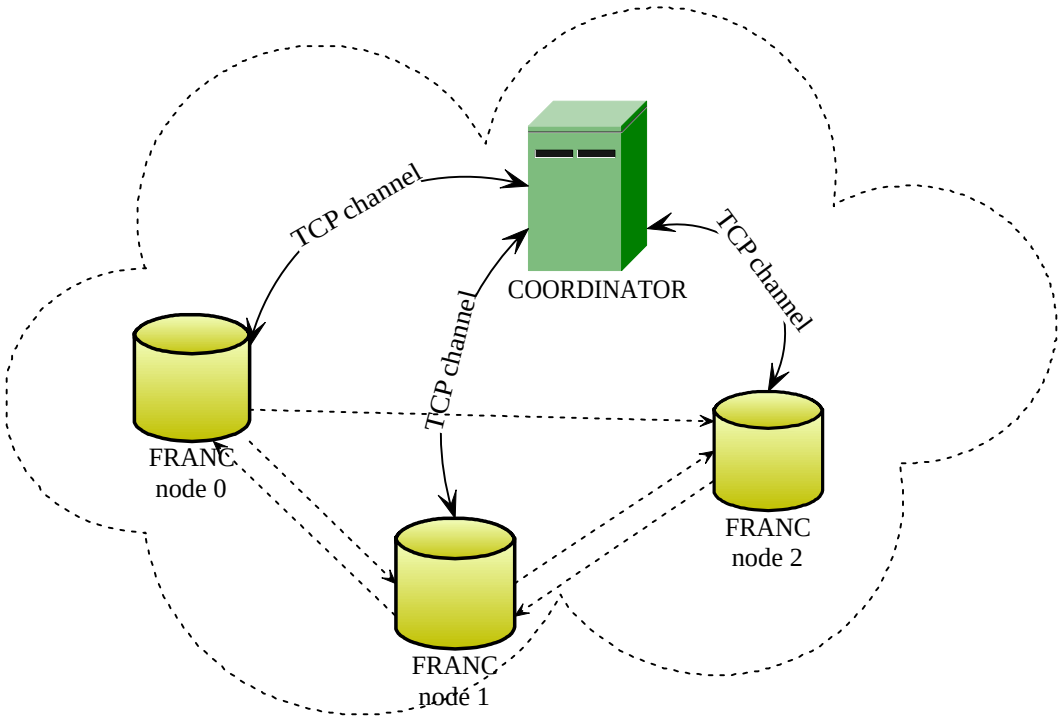

Figure 1: Simulation Overview

3 Architecture

The architecture of the simulator can be defined as a distributed decentralized architecture with a lightweight coordinator. The nodes communicate freely between themselves as if there was no simulation. The coordinator just controls the start and end of the simulation and collects the logs from the nodes. The network topology simulation is implemented directly as a layer for the nodes. Figure 1 reflects this architecture.

3.1 Coordinator

The Coordinator is limited to some well defined tasks, which do not make it a bottleneck for the simulation. Those tasks are:

- *Extract the simulation information from an XML configuration file for each node in the network topology.*
- *Wait for incoming connections from the nodes and sends the simulation information to each one of them.*

- *Announce the start time of the simulation process.*
- *Check periodically during the simulation process if the behavior is correct.*
- *Announce the end of the simulation process.*
- *Request the logging information of each node and store it for later use.*
- *Terminate each node and ends the simulation process.*

3.2 FRANC Simulation node

Each simulation node has a completely autonomous behavior based on the pre-configured information sent by the Coordinator. All the logic for receiving and interpreting this information is stored inside the node. The upper layers of a FRANC simulation node are unaware if the node is running in real or simulation mode. This is one of the advantages of simulation capabilities we implemented. The node's tasks can be summarized as follows:

- *Connect to the Coordinator.*
- *Receive its configuration information.*
- *Interpret and store the information in its context.*
- *Start its own simulation behavior at the start time indicated by the Coordinator.*
- *Send and receive messages to other nodes and logs its actions.*
- *End its simulation behavior.*
- *Wait for the Coordinator to announce the end of the simulation process.*
- *Check if the process has been successfully finished.*
- *Send its logging information.*
- *Terminate itself.*

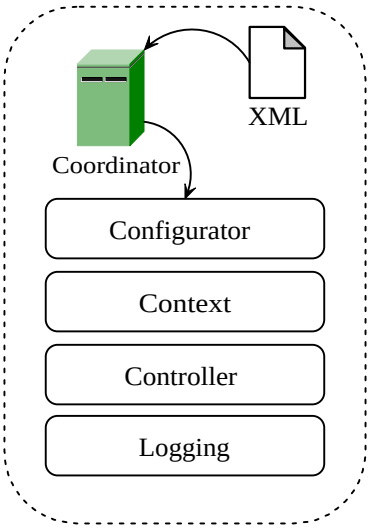

Figure 2: Simulator Architecture

4 Design and implementation

In the following section we discuss in more details the design and implementation issues encountered during the programming phase of the simulator using a top-down approach. We finish by enumerating the different packages we used to encapsulate the code and a summary of the coding conventions we used.

4.1 Coordinator

The Coordinator contains five different components. The idea behind these components is to minimize the coupling between multiple components, and thus provide a good modular design with minimum interaction between them.

4.1.1 Main

Description

This component contains the Coordinator main method. The component requires two command line parameters, each one in a XML format. The first one should contain the configuration of the simulation process and the second is a Log4j XML configuration file, according to the data type definition of Log4j[8].

The main method configures the simulation environment, initializes its own modules (server, context and constructor) and waits for incoming connections. Once

the entire network has been connected, the component checks periodically the state of the simulation according to its context and the state of the connections to different nodes in the predefined topology.

Implementation details

Here is the place to make a small reminder about the fact that the main method does not distribute and start the simulation nodes. The reasons for this are two:

1. Including this feature in the coordinator component would have coupled the application to a particular operating system, because of the usage of the Java Runtime.exec() method.

2. During testing of the Runtime.exec() method, we encountered weired behavior. The processes, launched using this method, were not executed until the main method had returned. Given the notes for this class described in the Java 1.4.2 API Specification, the user has no guarantee that the method starts correctly a process.

4.1.2 Server

Description

The role of the server component is to wait for connections from all simulation nodes, connect them, signal this to the Coordinator, and provide methods to the Coordinator to check if all connections behave in correct a way during the whole simulation process and to terminate the open connections upon signaling from the Coordinator.

Implementation details

This component has been implemented using the Singleton design pattern. The reason for this is that it represents inherently a unique object in the application. The server maintains TCP channels to all simulation nodes.

4.1.3 Context

Description

The role of the context is to provide a nice repository for storing the simulation information parsed by the Configurator. The repository is global for the Coordinator, thus any other component can access it.

Implementation details

In terms of implementation, the Context has been implemented using the Singleton design pattern. We consider it as inherently unique in the application. Its

internal structure is a HashTable, wrapped in a way to provide type checking functionality.

In order to insert an element in the Context, the developer should create a corresponding *ContextKey* and an object which extends the abstract class *AbstractConfig*.

4.1.4 Configurator

Description

This component is used to parse the XML configuration file for the simulation process. When an element is parsed, it is stored in the Context for later use by the application. For more information about how to create an XML simulation process, please refer to Section 6 on page 20.

Implementation details

The implementation of the Configurator uses JDom[9] as XML parsing technology. JDOM is considered to be the next Java standard for object-oriented XML parsing and it is likely to be included in the next Java API Specification (1.5).

4.1.5 Constructor

Description

The Constructor is a fairly simple component which takes the simulation information for a node and constructs an initialization event understood by the simulation node. This event is then sent to the simulation node by the Server.

Implementation details

The component is also implemented using the Singleton design pattern, because it is inherently unique to the application.

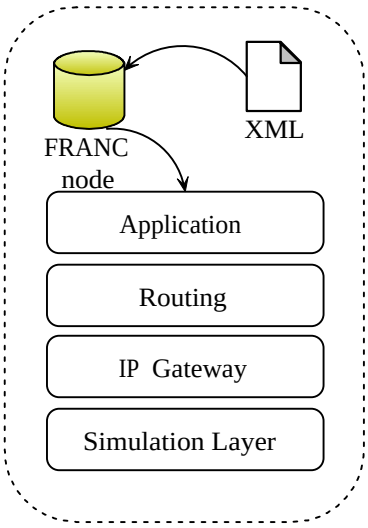

Figure 3: Simulation Node

4.2 Simulation node

A node in FRANC is composed of layers arranged as a stack. Two layers are mandatory: the data-link layer and the routing layer. The data-link layer is always the lowest since it is responsible for sending and receiving messages to and from the network. A simulation node is simply a FRANC node whose datalink layer has been replaced by a simulation layer (see figure 3). Since the simulation layer offers the same services to its upper layer as any other datalink layer, nothing has to be modified in upper layers to run a simulation. This was one of the requirements of the simulator.

4.2.1 Simulation layer

Description

The simulation layer's purpose is to emulate a wireless ad-hoc network. It replaces the data-link layer of a FRANC node and acts like a firewall. It must filter incoming and outgoing messages according to its current state (active or not, transmission quality, range and position) and add headers reflecting its position and range to outgoing messages. Its other task is the logging of events: schedule, position, range and quality updates and network traffic, of course.

Implementation details

The communication between the nodes uses UDP multicast, thus every node

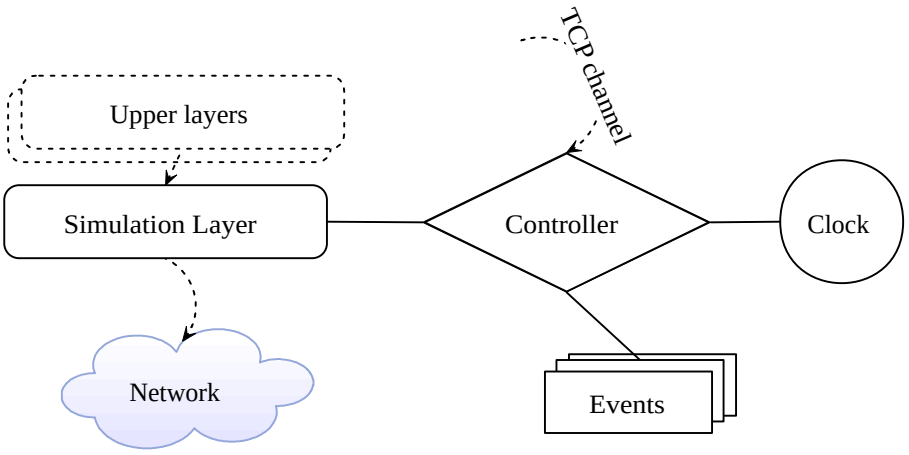

Figure 4: Simulation Layer Overview

receives every message. The simulation layer appends its range and its current position to every message it sends. When receiving a message it uses those appended parameters to determine if the message has to be dropped or not: it computes the distance between itself and the sender and compares it to the sender's range. Additionally it uses its transmission quality variable (a probability) and a random number to emulate transmission hazards.

4.2.2 Controller

Description

The Controller component was designed to control the simulation behavior of the node. It connects to the Simulator (TCP Channel in figure 4), gets the simulation information for the node it is managing and controls the behavior of the Simulation Layer described above according to this information.

Implementation details

In terms of implementation, the Controller always starts as a separate thread and blocks the whole node until it has successfully connected the node to the Simulator and received the configuration information. Then it starts the simulation process and unblocks at the same time the Simulation Layer. The Simulation Layer and the Controller represent the Observer design pattern. The Controller and the Clock also represent the Observer design pattern. Figure 4 gives a visual representation of these relationships.

4.2.3 Clock

Description

The Clock component was to provide a tick after a given interval of time. This is actually the speed of the simulation. Its default value is 1000 milliseconds. It can be changed in the XML configuration file of the simulation process. However, trying to increase a lot the simulation speed can have unpredictable results given the fact that the clock is implemented as a Java thread. All our tests were done at the default speed, and they showed an extremely stable precision.

Implementation details

To ensure a high level of accuracy, it gets the current time when it starts running, then issues a tick every 1000 milliseconds that pass, even if the notification does not occur every 1000 milliseconds exactly. Thus the notification that a time period has passed will perhaps not be very accurate, but the notification that some number of time periods have passed will be. In other words, the clock is accurate over long stretches. Given the vicissitudes of thread management, this is about the best that can be expected. All our tests show an excellent precision when running at speed 1000 milliseconds.

4.3 Simulation functionalities

As mentioned in subsection 4.2.1 on page 12, the emulation of an ad-hoc network topology is the role of the simulation layer. Let's have a closer look at the implementation of this feature.

4.3.1 Mathematical model

Description

We used the *Unit Disk Graph* model or *UDG*. Briefly, it means that a node A can communicate with a node B if and only if the distance between A and B is less than the transmission range R. In other words, each node has a circle of radius R and communication is possible only with nodes that fall within this circle. The physical behavior of radio waves is not taken into account, i.e. the transmission quality is not a function of the square of the distance. Since such a realism was not needed and neither wanted by our supervisors, *UDG* was an appropriate choice. However we completed the model by adding two other parameters. A probability for modeling some unreliability of the network and boolean variable to model the state of the node (active or inactive).

Implementation details

The model is implemented using five variables: x and y coordinates, transmission range, transmission quality and state (named schedule from now on). The position and schedule variables are updated using the event based mechanism described below, each time the controller's clock ticks.

4.3.2 Simulation events

Events can have two roles: be logged, update the node's state or both. The loggable events are those that carry useful information for the user from simulation point of view. This way we can control in a unified manner how that information is structured and presented to the user. Some of those events update the SimulationLayer's state and others do not. For example a new position event will clearly have an impact the layer's state, but an incoming message will not. The non-loggable events are mainly or internal use, for example checking the coordinator-controller connection.

4.4 Application components and directory structure

The application's implementation is divided in different modules which we call components. They are all encapsulated in packages. Most of the components are loose coupled in order to allow a nice overall modular design. Here are all the packages and their brief description:

ch.epfl.lsr.adhoc.simulator is the root of all the packages

.config contains the static context of the application. The context is a Java class representation of the simulation XML configuration file.

.controller contains the controller/clock implementation of the simulation layer component.

.events contains all different events used in the application, such as StartSimulationEvent, EndSimulationEvent, InitEvent, StopEvent, etc.

.layer contains the implementation of the Simulation layer conforming to the FRANC data-link layer convention [2].

.mobility contains the implementation of the functionalities of the simulator, such as node mobility, node time schedule, range and quality.

.modules contains the main simulator's components such as server, simulator context and constructor of simulation nodes from the context.

.testing contains most of our testing efforts on individual components.

.util contains different utility packages for XML handling, logging, NS2 and Java nested exceptions.

root/simulator/testbench contains two different simulation configuration examples, each one illustrating one particular functionality of the simulator. The first illustrates a time schedule behavior and the second one a node mobility pattern.

4.5 Coding Conventions

During the coding phase of the project we tried to comply to the Java coding standards used at CERN[4] and defined in [12]. Among all the conventions defined in this document we were particularly respectful of the following ones:

- **Line length**: usually not longer than 80 characters.
- **Prefixes**: members are prefixed with `m_`, parameters with `p_` and local variables are not prefixed.
- **Documentation**: classes, methods and members are thoroughly documented with Javadoc comments `/** */` whether they are public or private. This allows users of IDEs like Eclipse to see all the documentation online at all times, which saves time when editing the sources.
- **Comments**: `//` line comments are restricted to local variables (or task tags during development). We favor `/* */` block comments when documenting parts of the code.
- **File name prefixes**: abstract classes are prefixed `Abstract` and interfaces are prefixed with the single capital letter `I`.
- **Interfaces**: when the implementation is subject to future improvements or changes or when modularity is required, use interfaces to reduce coupling between classes.
- **Visibility**: members are private. Basically all methods are private. Their visibility is only augmented when they should provide a service to other classes (default or protected) or other packages (public).
- **Packages**: classes working together to provide a service, are grouped in a package. Classes providing different kinds of service are in different packages.

[4] European Organization for Nuclear Research

5 Testing strategy

5.1 Overview

Given the fact that the simulator's architecture has a modular design with an excellent loose coupling of the components, we proceeded in testing separately each component. This allowed us to ensure the wanted behavior before make full scale tests. After having tested all the components and their individual behavior, we proceeded in testing the simulator as a whole thanks to two different test configurations.

5.2 Organization

The following section presents the organization of our testing strategy.

- The package *ch.epfl.lsr.adhoc.simulator.testing* contains all tests on individual components
- The folder *testbench* contains the configuration of an entire simulation process

5.3 Tests on individual components

The following tests can be found in the package testing of our simulator:

SimpleTestLayer is a simple implementation of a FRANC Application layer which we used to evaluate and debug our own SimulationLayer. SimpleTestLayer is also used by the testbench in order to test the whole simulation process.

TestController tests the correct behavior of the SimulationLayer, SimulationController, and SimulationClock at the same time. In this test we wanted to make sure that the three components interacts well with each other.

TestFileTransfer tests the file transfer mechanism used to transfer logging information from simulation node to main simulator.

TestMobility tests the correct behavior of the data structures located in the .mobility package[5]. Dumps the content of the data structures for manual verification and prints for each node its position, every seconds from 0 to 1000.

TestMobilityParser tests the correct behavior of the NS2 parser component. It makes a dump of the mobility pattern for comparison.

[5] see section 4.4 on page 16 for a brief package description.

TestSimConfigurator tests the XML configuration of the simulator. It makes a dump of the simulation context after parsing and allows to compare it with the configuration file.

TestSimulatorControllerCom tests the communication between the Simulator and the simulation node. The communication is managed from the SimulatorServer and the SimulationController of the simulation node.

TestTCPServer tests a typical TPC server implementation (TCP queue, accept timeout, socket timeout, etc.)

5.4 Testbench

The testbench is a directory structure, a collection of configuration files and a script for automating the process of testing the simulator as a whole. It contains simple tests easy to understand and verify which were used to validate the final product.

There are currently two types of tests which enable us to ensure that the simulator behaves correctly. The first one tests the ability of a node to switch on and off at given times. Two nodes try to communicate but they are too far from each other to do so, thus they need a third node that works as a bridge. The on/off behavior is tested on the bridge and observed on the two other nodes (if their messages are delivered or not).

The second type is similar, but the accent is on the mobility. Our third node, which serves as a bridge between nodes 1 and 2, is always active, but travels between nodes 1 and 2. If it is too far away the two other nodes cannot communicate.

5.5 Coverage Analysis and Deadlock detection

Given the fact that our application is fully multi-threaded with a lot of shared variables between executing threads and particular meeting points of some threads, the need for a coverage analysis is completely justified in order to detect any possible deadlock and ensure that an important percentage of the code has been executed. Unfortunately we had no time to realize this idea, and hope addressing it during the second phase of the project, next semester.

6 Simulation guidelines

This section is intended to be a short manual for users of the simulator. It is not a developer's manual.

6.1 Configuration

6.1.1 FRANC

There's nothing particular to do for the simulator here. Just configure a FRANC node as described in [2]. We will refer to the corresponding configuration file as `franc.xml`.

6.1.2 Mobility

Mobility patterns must follow the format used for the NS2 simulator[6]. Such patterns can be written manually or generated automatically using the setdest[7] utility provided with NS2. Rather than giving the complete grammar of NS2's mobility patterns we prefer giving an explained example of the format accepted by our parser.

First of all a mobility file can contain as many comments as ones wishes. Of course, comments are ignored by our parser, as well as empty lines.

```
#A comment starts with # and stops at the end of the line
```

The file must begin with node definitions. A node definition defines the node's initial position along X, Y and Z axes, but the value on Z is always null. Three lines must be used, one for each axe. Here is an example of a file defining three nodes.
Note that the node IDs are in increasing order starting from zero.
One cannot define a node ID equal to one, if there is no node with ID equal to zero.

```
$node_(0) set X_ 380.825962687796
$node_(0) set Y_ 376.809106063166
$node_(0) set Z_ 0.000000000000

$node_(1) set X_ 961.976277430324
$node_(1) set Y_ 842.259984507709
$node_(1) set Z_ 0.000000000000
```

[6] see [3]
[7] in NS2's distribution 2.26, it is located under indep-utils/cmu-scen-gen/setdest

```
$node_(2) set X_ 781.539181377774
$node_(2) set Y_ 715.421886604222
$node_(2) set Z_ 0.000000000000
```

Mobility patterns follow, as an ordered sequence of orders.

```
$ns_ at 2.0000 "$node_(0) setdest 376.8583 552.9234 7.9263"
$ns_ at 2.0000 "$node_(1) setdest 135.0701 119.8773 7.1105"
$ns_ at 2.0000 "$node_(2) setdest 48.5678 128.8905 4.4397"
$ns_ at 24.2244 "$node_(0) setdest 376.8583 552.9234 0.0"
$ns_ at 26.2244 "$node_(0) setdest 465.0890 118.74282 0.0670"
$ns_ at 156.41959 "$node_(1) setdest 135.07012 119.8773 0.0"
$ns_ at 158.41959 "$node_(1) setdest 542.8177 303.2304 6.02286"
$ns_ at 213.44505 "$node_(2) setdest 48.5678 128.8905 0.0"
$ns_ at 215.44505 "$node_(2) setdest 129.0663 136.3148 9.6791"
```

Each line represents an order for a node. The first number represents the time at which the new order occurs. The number following `$node` naturally indicates the concerned node and the last three numbers indicate the new destination (as a two dimensional point) and the new speed. The lines are sorted by increasing order of their time field.

When generating a mobility file with NS2's setdest utility, a special node `god` may appear:

```
$god_ set-dist 1 2 1
$ns_ at 47.3787 "$god_ set-dist 0 1 2"
```

Such lines are ignored by our parser.

Important notice when writing a mobility file by hand

Pauses must be explicit: when a node reaches its destination, it must have an order that tells him to go elsewhere or to stay at his current location. Due to the structure of NS2's mobility patterns, it is hard for the parser to know for certain if there is an implicit pause or not. NS2's *setdest* only generates explicit pauses. You can find below the pattern for writing pauses. The writer of the file must compute himself the value of `someTimeT2`: `someTimeT2` = time value when the node reaches its destination (`someX`,`someY`).

```
$ns_ at someTimeT1 "$node_(0) setdest someX someY someSpeed"
...
$ns_ at someTimeT2 "$node_(0) setdest someX someY 0.0"
```

6.1.3 Simulator

Outline

The simulator's XML configuration file is divided in sections, each of them concerning a different group of options: *SimulationConfig*, *SimulationLayer* and *SimulationNetwork*, which itself contains the two subsections *global-config* and *custom-config*. Here is an outline of the file:

```
<Simulator>

  <SimulationConfig>  ...   </SimulationConfig>

  <SimulationLayer>   ...   </SimulationLayer>

  <SimulationNetwork ...>
    <global-config> ... </global-config>

    <custom-config>
      <node id="1" ...> ... </node>
      <node id="2" ...> ... </node>
      <node id="3" ...> ... </node>
    </custom-config>
  </SimulationNetwork>

</Simulator>
```

SimulationConfig

It contains some global parameters for the simulation and its distributedness. The first parameters concerns the server component of the main coordinator. These parameters allow to personalize the server TCP connection socket options. The ip and port fields indicate the port and ip of the listening server, the timeout specifies the connection timeout in milliseconds for each client trying to connect to the server and queue field indicates the number pending connections the server should queue. The clock parameter indicates the clock speed of the simulation process also in milliseconds. The duration field indicates the end time of simulation in milliseconds and the output-dir field gives the possibility to customize the output directory for storing all files generated by the simulator.

Example:

```
<SimulationConfig>
  <server ip="127.0.0.1" port="7666" timeout="20000" queue="20"/>
```

```
  <clock>1000</clock>
  <duration>30000</duration>
  <output-dir>logs</output-dir>
</SimulationConfig>
```

SimulationLayer

This section only specifies the SimulationLayer Java class. It is used to replace the data-link layer of a FRANC node. The corresponding *DataLinkLayer* parameters in `franc.xml` are replaced by the ones provided here. Unless multiple implementations for this class are provided, there's nothing worth modifying here.

Example:

```
<SimulationLayer>
  <name>SimulationLayer</name>
  <class>ch.epfl.lsr.adhoc.simulator.layer.SimulationLayer</class>
</SimulationLayer>
```

SimulationNetwork

The size of the network should be indicated in the field *size*.The second field allows you to specify the mode to be used by the simulator:

auto In this mode, you do not need to specify the network topology in *custom-config*. The simulator uses only the information defined in the *global-config*. This means that it will create a default network with *size* nodes and each node will inherit the configuration by the default specified in *global-config*.

manual In this mode, the user should specify the *custom-config* flag and for each node which should be different from a default node, specify the particular behavior.

Example:

```
<SimulationNetwork size="3" mode="manual">
  <global-config> ... </global-config>

  <custom-config>
    <node id="1" ...> ... </node>
    <node id="2" ...> ... </node>
    <node id="3" ...> ... </node>
  </custom-config>
</SimulationNetwork>
```

global-config
This field allows the user to provide default information for the network nodes:

node-ConfigIn User provides the default configuration file for a FRANC node according to the specification of FRANC.

node-ConfigOut The simulator modifies the file specified in the node-configIn field and creates a new file with the name provided in this field. Actually, this is the file used to start the node in simulation mode

node-errorLog User provides the name of the file that should be used to log error messages for a simulation node.

node-simulationLog User provides the name of the file which should be used to store simulation relevant information. Actually, this is the file to consider during the data processing of a given simulation process.

node-transmissionDefaults User should provide the default transmission range and transmission quality of a simulation node. The unit of the range could be anything, but it should be interpreted according to the units in the mobility pattern. If the speed of a node is in meters per second, the range should be considered in meters.

mobility-pattern User provides the mobility file (generated by NS2 tool setdest) and specifies the time scale to be used when interpreting the mobility pattern. This is needed because *setdest* specifies no units. The time scale has to be expressed in milliseconds. When time values are read from the mobility file, they are multiplied by the time scale. Thus a value of 1000 should be used when dealing with a mobility file expressed in seconds.

Example:

```
<global-config>
  <node-configIn>franc.xml</node-configIn>
  <node-configOut>franc_ready.xml</node-configOut>
  <node-errorLog>error.log</node-errorLog>
  <node-simulationLog>simulation.log</node-simulationLog>
  <node-transmissionDefaults range="100" quality="1"/>
  <mobility-pattern timeUnit="1000">mobility.txt</mobility-pattern>
</global-config>
```

custom-config

Thanks to this parameter the user could override the default behavior specified in the *global-config*. If some of the fields are not specified, they take automatically the default values in *global-config*. In the custom-config, the user could also specify an on/off schedule for a given node. The schedule unit is milliseconds and the field *scaleBy* saves you time when writing the milliseconds.

Example:

```
<custom-config>
  <node id="1" range="100" quality="1">
    <node-configIn>config/nodes/in/simpleFranc.xml</node-configIn>
    <node-configOut>config/nodes/out/00.xml</node-configOut>
    <node-schedule scaleBy="1000"> - scale by 1000 to get seconds
      <on atTime="0"/>   - at time 0 seconds
      <off atTime="10"/> - at time 10 seconds
      <on atTime="20"/>  - at time 20 seconds
    </node-schedule>
  </node>

  <node id="2" ...> ... </node>
  <node id="3" ...> ... </node>
</custom-config>
```

6.1.4 Log4j

The simulator provides two types of logging. The first one is only for the main simulator and can be freely configured through an XML file which has to be provided as a command line argument. This type of logging is used only inside the coordinator component of the simulator in order to log any error messages during the whole simulation process. Remember that the simulation process is supervised by the coordinator. The user is not supposed to modify this file, thus no documentation is provided here.[8] Just use the provided file. The second type of logging is inherent to the simulator and is stored in a special Java class. It should never be changed without carefully exploring the logging process of the simulator.

6.2 Logging

A simulation node logs two types of events: error and simulation events. The user could provide the storage files for these events in the *global-config* using fields

[8] if you are interested in Log4j, please refer to [7] and [8]

node-errorLog and *node-simulationLog*. Simulation events are logged in a unified manner using XML. A log entry begins with a header that contains the class name of the event and the simulation and system time at which the event occurred. The header is followed by the parameters specific to the event. Here is an example of a log entry:

```
<SimulationEvent
  name="ch.epfl.lsr.adhoc.simulator.events.NewLocationEvent"
  simTime="0"
  systemTime="Tue Feb 03 16:51:03 CET 2004">

    <param name=x>100.0</param>
    <param name=y>60.0</param>
</SimulationEvent>
```

6.3 Use cases

For those who did not bother reading the previous pages, here is a brief summary of what to do for running a simulation. If at any point you are lost, it is probably time for you to consider reading section 6 from the beginning, that is from page 20.

6.3.1 Only one type of node

Fairly easy! Let `myFranc.xml` be the configuration file of the nodes you want to simulate, let `myMobility.txt` be the mobility file you want to use, let N be the number of nodes you want to simulate, let RANGE be their transmission range and let T be the duration of the simulation (in milliseconds). Edit the simulator's configuration file as follows (only modified parameters are shown):

```
<Simulator>
  <SimulationConfig>
    <server ip="YOUR_IP_GOES_HERE" ... />
    <duration>T_VALUE_GOES_HERE</duration>
  </SimulationConfig>

  <SimulationNetwork size="N_VALUE_GOES_HERE" mode="auto">
  <global-config>
      <node-configIn>myFranc.xml</node-configIn>
      <node-configOut>myFranc_ready.xml</node-configOut>
      <node-errorLog>error.log</node-errorLog>
      <node-simulationLog>simulation.log</node-simulationLog>
```

```
      <node-transmissionDefaults
                  range="RANGE_VALUE_GOES_HERE" quality="1"/>
      <mobility-pattern timeUnit="1000">
                  myMobility.txt
      </mobility-pattern>
    </global-config>
  </SimulationNetwork>
</Simulator>
```

This configuration of simulation process will create a network of N nodes obeying the mobility pattern `myMobility.txt`. Each one of them uses default FRANC configuration `myFranc.xml`, logs error events in `error.log` and simulation events in `simulation.log`. They have all a default range of transmission RANGE and perfect quality of 1.

6.3.2 Multiple types of node

Not that difficult! Begin by doing the same as described above, then continue editing the configuration file with some more assumptions: all nodes are configured using the default values except nodes 23, 45 and 12 which have their own configuration files, transmission ranges and qualities. For example let's say that node 23 has a transmission range of 432, a transmission quality of 50%, uses mySpecialFranc.xml instead of the default myFranc.xml and is only active for 3 minutes. Here's what the configuration file would look like:

```
<SimulationNetwork ... mode="manual">
  <global-config> ... </global-config>

  <custom-config>
  <node id="23" range="432" quality="0.5">
    <node-configIn>mySpecialFranc.xml</node-configIn>
    <node-configOut>mySpecialFranc_ready.xml</node-configOut>
    <node-schedule scaleBy="1000">
      <on atTime="0"/>
      <off atTime="180"/>
    </node-schedule>
  </node>

  <node id="45" ...> ... </node>
  <node id="12" ...> ... </node>
</custom-config>
```

7 Limitations and Future development

In the following section we give an insight of the limitations of our application and enumerate some possible improvements. Here is the place to remind that some of the limitations would be addressed next semester after careful discussion with David Cavin and Yoav Sasson.

7.1 Limitations

Node's application behavior

Currently the simulator is able to simulate a network where each node has a different application behavior. However the actual architecture does not distinguish the application behavior of the nodes. When a node connects to the simulator, the simulator could not distinguish between a Router node and a Chat application node and distribute the mobility patterns accordingly. It could be done by starting each node sequentially which is not user friendly. We will address this limitation next semester and enhance the simulator with manual mapping between NS2 node numbers and FRANC node numbers.

FRANC based applications

The simulator was designed for FRANC based implementations of nodes, thus our application is not able to simulate other implementations.

Speed of simulation

The speed of the simulation clock is configurable in the XML configuration file. Its default value is fixed to 1000 milliseconds, and our tests show an excellent behavior of the simulator at this speed on a single machine. However a huge number of nodes running on a single machine is likely to penalize this behavior and alter the simulation results because of the vicissitudes of the current Java thread management implementation. We remind that one simulation node is run in a separate Java Virtual Machine. Our advice is to keep the simulation speed reasonable.

Clock synchronization in a distributed environment

If the simulation process is deployed in a cluster of machines, the user should check the clock synchronization between the machines. The simulator assumes a good clock synchronization based for example on NTP[11].

Limited functionalities

Currently the simulator could simulate nodes with NS2[3] mobility pattern. It

could also simulate nodes which switch on and off their transmission during the time of simulation. It could simulate different transmission range for each node as well as different trasmission quality. However the quality of the link is only implemented with a probability of transmission, the physical model of the propagation of waves is not taken into consideration.

TCP connection failure
Currently the simulator would detect when a node TCP connection has died, and signal this in its log file. However the node is not implemented to try to reconnect to the server when it finds its connection dead.

7.2 Future Developments

Enhance the distributivity and functionalities
This improvement should make the whole distributed process user friendly and allow a nice distinction between different nodes. The idea behind this is not to connect the FRANC simulation node directly to the server of the simulator, but to delegate this task to an external client on the remote machine. We hope addressing this issue next semester.

Logging architecture
The logging architecture for simulation events should be reviewed, and a service for automatically retrieving upper layers' applications log files should be provided. We hope addressing this issue next semester.

Coverage analysis
Use a coverage analysis tool in order to test each line of code. We hope address this issue next semester.

Build a Graphical User Interface
Building a GUI for the simulator could be an interesting semester project for a student in Computer Science or Communication Systems.

Logs interpretation
This might be also a very good semester project. Interpret the logs generated by the simulator and generate different statistics in order to evaluate the performance of applications or routing algorithms.

8 Lessons learned

Development

During the development phase of the project we learned the following:

- The time spent at the beginning in order to make a nice, clear and modular design for the application was definitely not lost. It reduced a lot our efforts when putting all pieces of the application together.
- Even with a good starting design model, some important changes were made when we started the implementation of the application. Some of them ended in changing many times the behavior of some components.
- Never fear refactoring code was an excellent idea. It allowed us too improve the design of the application during development and make it even more modular then at the beginning. Many ideas came during implementation and with careful refactoring we succeeded in implementing them.
- The use of an Integrated Development Tool was definitely a must. We do not think we could have succeeded without it.
- Discussions in a team are very helpful and we all appreciated them.

Testing

During the quality testing of our application we learned some important lessons:

- The modular design provided the possibility to test each component separately. This was of particular importance when testing the application at the end. We think this approach saved us a lot of time when trying to put all the components together.
- A multi-threaded application has some particular issues (mutual exclusion of shared variables, synchronization and deadlock vulnerabilities). We encountered a lot of problems during our testing efforts on the testbench which were not detected while testing the individual components, such as deadlocks. This was particularly difficult to debug because of the complex behavior of the simulator and the fact that a deadlock occurs not very often. We had to draw deadlock graphs and look for a solution of the problem. Most of solutions involved slight changes of the design.
- Testing is definitely not a trivial task. It is not easy for the developers of an application to imagine crash scenarios. We think we were quite vigilant on this. Being two on the project helped to save time when one of us had problems.

- Testing is definitely not a passionate task. It takes a lot of time to imagine what to test after having done some basic tests.

9 Conclusion

During the development process, together with our supervisors we kept the design and implementation requirements high, not fearing the amount of time it would take to realize them. We put more emphasis on designing and implementing a good architecture than trying to finish everything by the end of the semester. We think we successfully designed a robust, modular and maintainable simulator for FRANC, integrated new tools and technologies, and did all this in a strong team spirit.

References

[1] FRANC: A lightweight Java Framework for Wireless Multihop Communication
AUTHOR: *David Cavin, Yoav Sasson, André Schiper*
http://lsrwww.epfl.ch/ip9

[2] Semester project: MANET Framework
AUTHOR: *Javier Bonny, Urs Hunkeler*
http://lsrwww.epfl.ch/ip9

[3] The Network Simulator NS-2
http://www.isi.edu/nsnam/ns/

[4] JEmu: A Real-Time Emulation System for Mobile Ad-Hoc Networks
AUTHOR: *Juan Flynn, Hitesh Tewari, DOnald O'Manhony*
http://www.cs.tcd.ie/omahony/jemu-iei.pdf

[5] Effective Java, Programming Language Guide
AUTHOR: *Joshua Bloch*

[6] Java Performance Tuning, Second edition
AUTHOR: *Jack Shirazi*

[7] The Log4j project
http://logging.apache.org/log4j/docs/

[8] The complete manual: Log4j
AUTHOR: *Ceki Gülcü*

[9] JDOM : a complete, Java-based solution for accessing, manipulating, and outputting XML data from Java code
http://www.jdom.org/

[10] Eclipse IDE
http://www.eclipse.org/

[11] Network Time Protocol (ntp)
http://www.ietf.org/rfc/rfc0958.txt

[12] Java Code Conventions for use in EDH
http://ais.cern.ch/apps/edh/CodingStandards

[13] The Not So Short Introduction to LaTeX 2_ε
AUTHOR: *Tobias Oetiker*