Semester Thesis

Development and Testing Layer for Ad-Hoc Network Software in Java

Yves Weber
webery@student.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Winter 2003/04

Prof. Dr. Roger Wattenhofer
Distributed Computing Group
Advisor: Aaron Zollinger

Contents

1 Introduction **2**

2 Design of the Emulator **3**
2.1 The Four Main Parts . . . 3
2.2 Replacing `MulticastSocket` . . . 4

3 Implementation: Class Overview **5**
3.1 Core Classes . . . 5
3.2 The Replaced Socket . . . 6
3.3 GUI Components . . . 6
3.3.1 DrawPanel . . . 7
3.3.2 ControlPanel . . . 8
3.3.3 Output . . . 8

4 Outlook **10**
4.1 Open Problems . . . 10
4.1.1 Stopping Clients . . . 10
4.1.2 Mapping a Socket to its Client . . . 10
4.1.3 Clients Calling `System.exit()` . . . 11
4.1.4 Options of `DatagramSocketImpl` . . . 11
4.2 Possible Extensions . . . 11

5 Conclusion **13**
5.1 The Program . . . 13
5.2 Personal Experience . . . 13

A Short Manual **14**

Chapter 1

Introduction

In the exercises of the Mobile Computing lecture [1], the task was to implement an Instant Messenger (IM) for an ad-hoc network. The messenger should communicate through `java.net.MulticastSocket`s and use multihop routing to allow interaction between two clients which do not hear each other directly. While the implementation was very interesting, it showed that testing the IM in the real world is not an easy task.

The simple approach of taking a number of notebooks with wireless equipment running in ad-hoc mode and starting an instance of the IM on each of them has some disadvantages:

- It is difficult to test the IM on a specific graph, e.g. on a circle. A lot of coordination is needed to ensure that each wireless card is in the transmission range of exactly two others.
- When the multihop routing algorithm fails, it is hard to find out whether it failed because of packet losses or because of the algorithm itself, i.e. it is not possible to test the IM in a perfect, deterministic environment.

The task of this semester thesis was to develop a testing layer which allows to start several instances of the IM on one computer. The layer takes as its input the description of a graph, where each node represents an instance of the IM and the existence of a link between two nodes means that the two corresponding instances hear each other. The layer then emulates the ad-hoc network by forwarding the messages from one IM to another if the graph allows this.

The emulator[1] should be completely transparent to the IM, i.e. the IM cannot recognize if it is running on top of the layer or in a real ad-hoc network. This implies that the code of the IM does not need to be changed to make it compatible to the testing layer. To improve the versatility of the ad-hoc layer, its compatibility should not be restricted to the IM done in the Mobile Computing lecture. It should be able to collaborate with every Java program that uses `MulticastSocket`s to communicate.

[1] In the following document, the development and testing layer is also called emulator. Programs like the IM that use the layer are called clients; each client corresponds to one node of the graph representing the ad-hoc network.

Chapter 2

Design of the Emulator

This chapter contains the main design ideas behind the emulator. After a short overview on the most important modules of the emulator, information is given about how the emulator replaces the socket used by a client without the need to adjust the client.

2.1 The Four Main Parts

The project consists of the following four main parts:

- A data structure to represent the underlying graph of the emulator. It abstracts from the fact that it represents an ad-hoc network, i.e. it is just a collection of nodes with some links between them. It should basically offer methods to add and remove nodes and links as well as the possibility to read the whole graph from or write it to a file, respectively. The required methods of this data structure are defined in the interface `Graph`.
- The emulated `MulticastSockets`. These sockets must have the same interface as the `java.net.MulticastSocket`, but instead of sending the UDP packets over a network, they are passed directly from one socket to another since all instances of the socket are running on the same machine. The sockets abstract from the program they belong to and they do not know anything about the underlying graph structure. All they can do is sending and receiving packets of type `java.net.DatagramPacket`.
- The GUI. Besides the parts that take care of the emulation itself, much emphasis was put on a convenient user interface. The user should always see what is currently going on and have the possibility to change as many parameters as possible at runtime.
- The core. This part represented by the class `Emulator` is holding everything together. It keeps a list of all open sockets and a reference to the GUI. When a packet needs to be sent, this part checks from which client it is coming from and passes the packet via the link to all neighboring clients.

The idea was to connect all those four parts by a small interface and keep each of them independent of the implementation of the other ones. However first versions of the emulator have shown that this approach produces an unreasonable amount of traffic between the data structure representing the graph and the GUI since most actions of the graph (including sending a packet and thus increasing a counter on the corresponding link) required an update of the user interface. This is the reason why the graph data structure is now part of the GUI (namely the class `DrawPanel`), thus partially sacrificing the independence between the graph and the GUI to increase efficiency.

2.2 Replacing `MulticastSocket`

A normal client usually opens one or more `java.net.MulticastSocket`s and starts sending and receiving UDP packets. A very challenging task was to replace this socket without the client knowing about it, i.e. without changing the code of the client. It took three attempts to finally succeed:

1. The first approach was to just write a new socket class with the same interface as `java.net.MulticastSocket`. Of course, this worked very well, but it had a big drawback: The user needed to adjust the code of the client in at least two locations: an `import` statement at the beginning to import the modified socket, and at the location the socket is created, the user had to call a different constructor.

2. The second idea was to use a `ClassLoader` to solve the problem: When the client creates a new instance of `MulticastSocket`, the default `ClassLoader` receives a request for this class and returns the default `MulticastSocket`. By replacing the default `ClassLoader` with a personalized one, this request can be intercepted and the modified socket can be returned. This procedure works until the point where the `ClassLoader` needs to register the modified socket to the system. For security reasons, the runtime system does not allow to register customized classes with a name starting with `"java."`, e.g. `java.net.MulticastSocket`. There is no way to circumvent this restriction[1].

3. After running out of ideas, the problem was posted in the Java Technology Forums [3]. Already the first reply [4] pointed out that the `DatagramSocket` class (which is the superclass of `MulticastSocket`) has the static method `setDatagramSocketImplFactory` to set a customized factory for creating the sockets. Using this method, the customized socket can be returned to the client without any changes, since the `DatagramSocketImplFactory` can be set by the emulator before any client is started. This finally allowed to replace the `MulticastSocket` in a manner totally transparent to any client.

[1]Actually, there is a way to circumvent this problem. The idea is to temporarily corrupt the internal string table of the runtime system so that the check `name.startsWith("java.")` in the method `defineClass()` of the class `ClassLoader` returns `false` even though `name` is the String `java.net.MulticastSocket`. The precise procedure to do this is described on [2]. But this approach is considered a dirty hack and heavily dependent of the current implementation on the Java runtime system.

Chapter 3

Implementation: Class Overview

In this chapter, an overview is given about all classes written for this project. Their tasks are described and a rough outline of their implementation is given. For more details and the description of all methods and parameters, the comments in the code of the classes should be consulted.

3.1 Core Classes

As mentioned in section 2.1, the most important class of the project is the class `Emulator`. It keeps all other parts together and gets everything to work. This is also where the `main` method of the whole project is located. The two principle tasks of the class `Emulator`, besides offering communication between the other parts, are:

- Starting the client applications. This is done with the help of the class `ClientThread`. To allow the user to run different applications in each node, `ClientThread` uses Reflection (`java.lang.reflect`) to load the classes specified by the user at runtime. Each client is started in a different `ThreadGroup` with the name of the corresponding node. This is required by the method `mapThreadToNodename()` to identify which thread belongs to which client[1].
- Distributing a packet between the sockets. As stated in section 2.1, a packet sent by a socket is passed to the class `Emulator`. Its task is to find all other sockets connected to the sender socket which are in the multicast group the packet is sent to. This is done by passing the packet to all links (class `Link`) of the sender. The links then check whether or not there is at least one socket of the desired multicast group on the other side and whether the packet gets lost by a simulated transmission error.

Since `Emulator` is implemented as a singleton, all classes can always access it using the static method `Emulator.getRef()`.

[1]see section 4.1.2 for restrictions of this approach.

3.2 The Replaced Socket

A very important class is EmuSocket. As described in 2.2, this class replaces the original java.net.MulticastSocket. All packets sent via this socket are passed to the Emulator class for further distribution instead of sending them out through a network adapter.

When the EmuSocket receives a packet, it cannot be directly forwarded to the program the socket belongs to. Instead, the packet has to be saved until the owner calls the blocking method receive(DatagramPacket). All packets waiting in the socket ready for being received are stored in a priority queue sorted according to their delivery time. This is necessary since the packets have different delays if the user changes the delay of a link during runtime.

Besides the two methods to insert a packet into the priority queue, all methods of class EmuSocket are inherited from the abstract class DatagramSocketImpl. An annoying aspect is that this abstract class has two methods—setTTL() and getTTL()—which are deprecated in the current Java API version 1.4.2. Not implementing these methods leads the compiler to generate an error message saying that the class EmuSocket should be declared abstract; but implementing the two methods results in a warning that the class is using deprecated methods. No way was found to circumvent this warning.

3.3 GUI Components

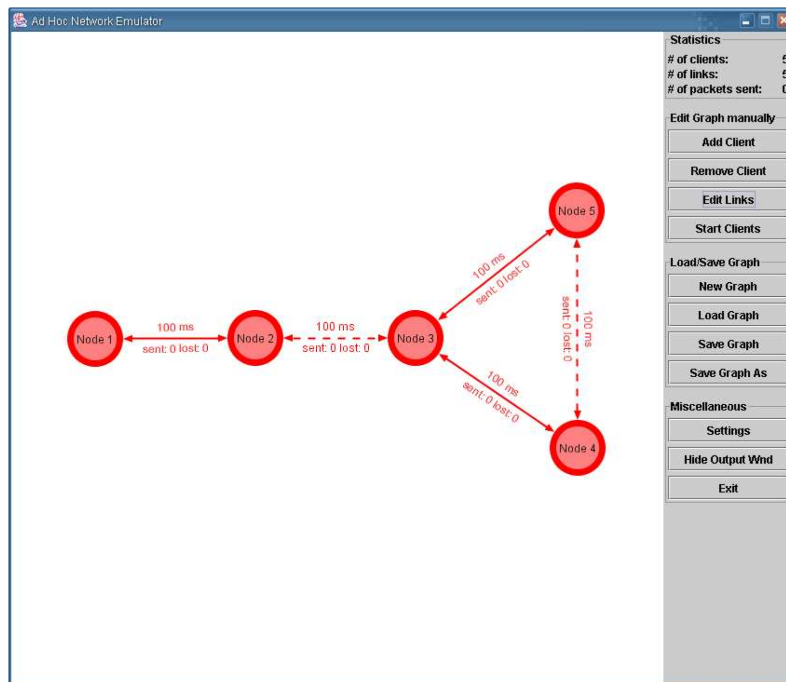

Figure 3.1: The main window of the Emulator with the DrawPanel on the left and the ControlPanel on the right.

Figure 3.1 shows the GUI of the emulator. This window is represented by the class `MainWindow`, which consists of two panels, namely the `DrawPanel` on the left to display and edit the underlying graph of the emulator and `ControlPanel` on the right. Figure 3.2 shows another important part of the GUI. It is a window where the output of all running clients goes to. This window is represented by the class `OutputWindow`. These three classes including their helper classes will be described in the next chapters.

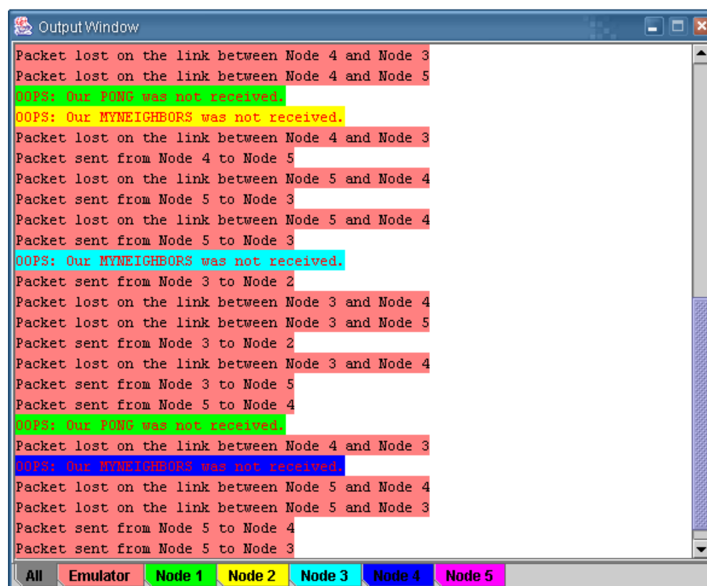

Figure 3.2: The output window collecting output from all running clients and the messages of the Emulator

3.3.1 DrawPanel

In the `DrawPanel`, the user can specify what the graph emulated by the ad-hoc layer should look like. Using the left mouse button, nodes can be moved around and links can be created and edited. A click with the right mouse button on a node brings up a window to edit (`EditClientWindow`) the node or creates a new node when clicked on the background. The implementation of the `DrawPanel` is pretty straightforward: The class extends `JPanel` and keeps a `Vector` of all currently available nodes (class `Client`) and links (class `Link`). A request to draw the panel is redirected to all these elements resulting in each element being drawn. When the `MouseListener` reports any clicks from the user, the `Draw-Panel` checks whether a node or a link was hit and generates the appropriate reaction, e.g. moving a client or creating an instance of `EditClientWindow`.

The `DrawPanel` also implements the interface `Graph`. Using this interface, other classes can get information about the current graph or even edit it, as described in section 2.1. The class `GraphFileWriter` is able to write the current graph to a file, while a complete graph can be read from a file using class `Graph-`

FileReader. So the DrawPanel is not only responsible for drawing the graph but also the data structure containing the graph.

3.3.2 ControlPanel

The ControlPanel contains no code other than for creating and positioning its buttons and a mouse listener to react to clicks on any of the buttons. The main function of the ControlPanel is to provide the user with another method to edit the graph. The possibilities to change the graph go far beyond those that are offered by the DrawPanel: The user can load and save complete graphs, start clients, remove clients[2] and change the properties of a link. This is done by creating and starting an instance of the appropriate window. The classes representing these windows have fairly self-explanatory names (e.g. EditLinkWindow or EditLinkPropWindow) and their implementation is—besides the creation and positioning of many SWING components—very short and easy to understand, consequently they do not deserve further explanation here.

Another job of the ControlPanel is to give the user the possibility to change certain global parameters of the emulator. Such parameters include the size of the buffer in which the sockets save the packets waiting to be received or the amount of milliseconds the arrow of a link flashes if the link was used to send a packet. These settings are saved as static fields in the class Options. To edit them, the ControlPanel opens an OptionsWindow. When the emulator is closed, the settings are saved in the file "emulator.txt" and reloaded when the emulator is started the next time.

The final task of the ControlPanel is to show some statistics of the running emulation process (e.g. how many packets have been sent) and keeping this information up-to-date. This is done by a private thread which renews the statistics once per second.

3.3.3 Output

The user can choose in the OptionWindow to redirect all output of the running clients not to the console but to a special window. This window is represented by the class OutputWindow. It consists of a JTabbedPane with one tab for each running client and the two additional tabs "Emulator" and "All". In the tabs of the clients, all the output generated by the corresponding client is shown. The text sent to System.out is printed in black, text to System.err in red. This way, the user is able to see a difference between the two streams. Output generated by the emulator itself (e.g. when a packet is lost due to a buffer overflow in a socket) is of course shown in the tab "Emulator". The last tab labeled "All" shows the output of all clients together, including messages from the Emulator. Different background colors are used, one for each client. The implementation of OutputWindow is quite straightforward: There are methods to add and remove tabs and a method to write text. The latter takes as parameters the name of the destination tab, the text to write, and a boolean flag whether or not the text is coming from System.err.

The main work—intercepting the print() and println() calls and redirecting them to the corresponding tab—is done by the class OutputHandler.

[2] see section 4.1.1 for restrictions when removing clients.

When started, it replaces the default streams with its own streams by calling `System.setOut()` and `System.setErr()` respectively. These streams have to find out from which client the text is coming. This is achieved by calling `Emulator.mapThreadToNodename()` with the current thread as its parameter. The `Emulator` class recognizes which thread belongs to which thread group; since the thread groups carry the names of their corresponding clients, this translation can be done easily. As stated in section 4.1.2, this mapping may fail. If the text cannot be associated with a client, it is printed with the same color as the messages from the emulator.

The user can choose not to use the `OutputWindow` but to print everything to the console. The `OutputHandler` then redirects the output to the default `System.out` and—if chosen by the user—prefixes the name of the sender to each line of text.

Chapter 4

Outlook

4.1 Open Problems

There are some problems for which no satisfying solution was found during this thesis. This was either because there was not enough time—or the amount of time needed to solve the problem was not justified—or because there appears to be no feasible way to solve the problem.

4.1.1 Stopping Clients

In the current version, removing a client from the emulation is done by removing the client from the graph, i.e. removing all links connected to the corresponding node and the node itself. The client itself keeps running. There are two possible approaches to change this:

- The client itself offers a method `externalStop()`, which allows the emulator to force a client to terminate. But this approach contradicts the universality of the clients. Further, this method has to respect some rules, e.g. not to use `System.exit()` (see section 4.1.3)
- The emulator stops the client without informing it. This could be achieved by identifying all associated threads and stop each one of them, or even simpler by stopping the whole `ThreadGroup` of the client. But since `Thread.stop()` and `ThreadGroup.stop()` are deprecated, this can not be done with the current JDK version.

4.1.2 Mapping a Socket to its Client

The current approach to use `ThreadGroup`s to identify which socket belongs to which client has a major disadvantage: If a socket is created as a reaction to an event from the GUI of a client, e.g. an `ActionEvent` when the user clicked a button, the mapping from the socket to its client fails. The reason for this is the fact that the code of the listener of such an event is executed by the AWT event dispatcher thread. This thread is the same for all clients and does not belong to a specific thread group.

If a socket is created and cannot be assigned to a client by the emulator, the socket is ignored by the emulator, and an error message is displayed to the

user. It tells him or her not to create a socket as a reaction to an event from the GUI but to instantiate all sockets when the client is started.

4.1.3 Clients Calling `System.exit()`

A major problem is that when the user manually wants to remove a client by closing it, it is very likely that the client calls `System.exit()` to terminate. Unfortunately, this also stops all other clients and the emulator itself. Possibilities to solve this problem are:

- A `SecurityManager`. Before the `System.exit()` call is executed, the runtime environment calls `checkExit()` of the current `SecurityManager`, if there is one installed. Unfortunately, installing a `SecurityManager` is not the perfect solution. It is very likely that the program which called `System.exit()` is left in an undefined state and is still running. So this approach leads to new problems.
- All clients could be run in different virtual machines. A `System.exit()` call would then only terminate the virtual machine of the corresponding client. But this makes communication between the clients much more complicated; this would go beyond the scope of this thesis.

The current implementation just assumes that either the clients can be terminated without the use of `System.exit()` or that the user does not try to close a client.

4.1.4 Options of `DatagramSocketImpl`

The interface `DatagramSocketImpl` implements the interface `java.net.SocketOptions`. This interface consists of the two methods `setOption()` and `getOption()` to set several options of the socket. Since most of these options change the behavior of lower levels which are not simulated by the emulator, most calls to `setOption()` are ignored. The only option ID having an effect is `SocketOptions.SO_TIMEOUT` to set a timeout of a call to `MulticastSocket.receive()`. Because all other option IDs have no effect, clients depending on them might show unexpected behavior.

4.2 Possible Extensions

During the work on the program, several ideas for additional features were proposed. While most of them are implemented, some had to be ignored because of time limitations. These possible extensions are listed in the following:

- Experiments have shown that running more than about a dozen clients in the emulator is not feasible since the user tends to lose track. There are much likely memory issues too, depending on the client programs. This is the reason why it would be very useful to run the emulator on several machines and emulate one big network on them. To implement this feature, precise coordination and synchronization between the instances of the emulator is needed to ensure consistency.

- At the moment, the user has the possibility to define the delay and the error probability or a deterministic error pattern for each link. It could be useful to set more precise properties. Such attributes could be:

  - Delay based on a probability distribution. This would imply that the clients could not rely on the FIFO property of the links any more.
  - Packet loss dependent on different factors, such as link load, node buffer status or packet length.
  - Probability for bit errors in a packet.

  The main challenge would be to integrate these features without making the link properties dialog window overloaded and still easy to handle.

Chapter 5

Conclusion

5.1 The Program

The current version of the testing layer is capable of emulating an ad-hoc network so that this program can be used as a tool to develop and test new ad-hoc software. The main goal—the nonnecessity to do any adjustments to this software to make it compatible with the layer—was mostly achieved[1]. Tests have shown that this emulator is capable of running up to 8 instances of the given Instant Messenger—a total number of 16 sockets—without any problem. Beyond that number, `OutOfMemoryExceptions` started to occur. Using a smaller client which requires less memory, the emulator succeeded to run more than 30 instances in parallel. The biggest open problem remaining is that there are limitations when a user wants to stop a running client either by removing it from the emulation itself or by closing it manually. But since there is barely a scenario where stopping a client is really essential, this is totally acceptable.

5.2 Personal Experience

It was very exciting to work on this project from the design over the implementation to the final testing phase. It was surprising how many aspects such as Reflection, Classloaders or Security Managers were needed to work with during the whole process, even though this thesis not look as if it would require such in-depth Java knowledge at the beginning.

Of course, there were also some frustrating setbacks, the biggest one being to realize that the approach to use Classloaders to replace the `MulticastSocket` would not lead to success, after a considerable amount of work on them. But the fact that another approach finally solved the problem in an even more elegant way was more than rewarding for the time lost before.

[1]See section 4.1.2 for information about the only restriction to the client.

Appendix A

Short Manual

Starting the Emulator The emulator is started from the console with the command `java -jar Emulator.jar`. It can also be started by double-clicking the jar file if this function is supported by the operating system. It is important that the programs to be run in a node are in the same directory as the emulator. If not, the runtime system is not able to locate the classes. Of course, they must be already compiled.

Editing the graph On the left side of the main window of the emulator, the graph can be edited. With the right mouse button clicked in an empty location, a new node can be added, while a right click on an existing node opens a window to edit its properties. The left mouse button is used to change the position of a node and to add or remove links. A node can be moved around by dragging it in the middle. When clicked on its border, a new link starting at this node is created. To remove a link, it must be grabbed at one of its arrows and released in an empty area of the graph.

The graph can also be edited by the use of the buttons on the right side of the window. These buttons offer even more possibilities to change the graph, e.g. adjusting the properties like the delay or the error probability of a link. A graph can also be saved or loaded using the corresponding buttons.

Starting the Emulation Process When the graph is drawn and the class name including the parameters for `main()` of each node is specified, the emulation process can be started by pressing the "Start Clients" button. Even when the programs are started, the graph can still be edited and new clients can be added. A subsequent click on the "Start Clients" button activates the recently added programs.

Stopping a Client To stop a client, the "Remove Client" button should be used. The emulator then removes all links of the corresponding node and closes all sockets of the client. The window of the client remains visible. If this window is closed, it is very likely that the client executes `System.exit()`. This would result in the termination of the whole emulator. This is the reason why preferably the "Remove Client" button should be used to directly close the window of the client.

Bibliography

[1] Homepage of the Mobile Computing course, summer 2003
`http://dcg.ethz.ch/lectures/ss03/mobicomp/index.html`

[2] Insane Strings (changing hardcoded strings at runtime)
`http://www.smotricz.com/kabutz/Issue014.html`

[3] Java Technology Forums
`http://forum.java.sun.com/`

[4] Java Technology Forums - Posting containing the advice to use a factory to replace the `MulticastSocket`
`http://forum.java.sun.com/thread.jsp?forum=4&thread=469427`