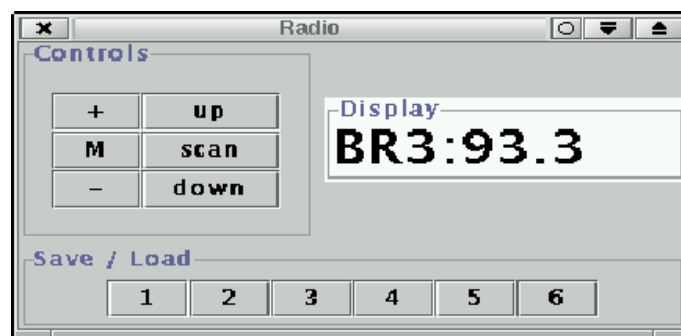


Assignment 3/ Object Oriented Systems Design – Sockets and RMI

Kenan Esau

November 2000



Tutor: Frank Müller
Course: M.Sc Distributed Systems Engineering
Lecturer: Mr. Prowse

Contents

1	Introduction	3
2	Sockets in Java	3
2.1	Sockets using TCP	3
2.1.1	Server	4
2.1.2	Client	4
2.2	Sockets using UDP	4
2.2.1	The Time-Server	5
2.2.2	The Time-Server-Client	6
2.3	Multicast	6
3	Remote Method Invocation	7
3.1	The Radio Simulator	8
3.2	Changes in the Design	8
3.2.1	The Model	9
3.2.2	The View	10
3.2.3	The Controller	11
4	Conclusions	12

1 Introduction

This assignment deals with two quite large topics:

1. Sockets in Java
2. Remote Method Invocation

According to this the assignment has two main parts. The first part is about sockets in Java. In this part the different approaches with sockets are described using the little examples given in the classes. The second part – dealing with RMI – describes the changes needed for the radio-simulator to introduce RMI and network capability.

The part dealing with sockets is a little bit shorter so I was able to focus a little bit more on RMI and the changes needed in the radio-simulator. Since this assignment is limited to about 2500 words it will be impossible to cover every topic in detail without breaking this limit.

2 Sockets in Java

Sockets can be used for communication between two programs. You can think of a socket as a bidirectional, network-capable pipe. In contrast to C in Java there is some functionality hidden to the user (programmer). In C the programmer has to explicitly call `bind()` – in Java this is hidden within the constructor of the appropriate Socket-class. With this simple Java-approach socket-programming is even easier than with C.

The socket-mechanism is very old they were first introduced in the BSD-Unix System Version 4.1cBSD [6]. One very old and well known program which uses sockets for network-transparent communication between client and server is the X-Window-system which was first developed at the MIT around 1987. All high level protocols like HTTP, SMTP, and many others rely on the socket mechanism.

2.1 Sockets using TCP

Sockets using TCP differ a little bit from the datagram-oriented sockets which are described in section 2.2 since they are stream-oriented. Before any communication can take place a connection has to be established and after the end of communication the connection has to be released again. Therefore connection-oriented Sockets are a little bit more complicated than connection-less but connection-orientation offers a lot more safety. Since every IP-packet has to be acknowledged every loss of a packet is noticed immediately.

2.2 Sockets using UDP

2.1.1 Server

A TCP-server-socket can be created by creating an instance of the class `ServerSocket`. After the server-program created a socket it “listens” to it for incoming requests. There are the following three steps which are executed after a new `Socket` is created:

1. The program is bound to the socket.
2. The server listens to the socket.
3. The server accepts connections to the socket.

In a C-program these three steps have to be executed explicitly by the programmer. If you are using Java these three steps are executed within the constructor of the socket class. This way a lot of details are hidden to the programmer but it makes programming a little bit easier.

For more details have a look at the classes `Sioux` and `Worker` which implement a very primitive “Web server”.

2.1.2 Client

The client works very simple too. The first step for the client is to create the socket. There for it has to know the IP-address or the name of the server and the port it wants to connect to. In the example below the port number is fixed and the server name can be set by a command line parameter.

```
Socket socket ;
socket = new Socket (server, 1777);
System.out.println("Client_" + clientCount + "_created_\
                    connection");

BufferedReader reader = new BufferedReader( new \
        InputStreamReader(socket.getInputStream()));
while (!reader.ready()); //Wait until reader is ready
while (reader.ready()){
    String answer = reader.readLine();
    System.out.println(answer);
}
socket.close();
```

2.2 Sockets using UDP

Sockets which are using UDP are connection-less since UDP is connection-less. In contrast to the sockets using TCP the sockets using UDP are datagram oriented. The UDP protocol is a lot simpler than the TCP protocol. Since there is no automatic ACK/NAK-mechanism like in TCP

2.2 Sockets using UDP

UDP is also a lot more insecure than TCP. But for a LAN and for small packets the UDP-approach is sufficient and the error rate is small enough. Since the missing of an ACK/NAK mechanism the loss of a packet can not be detected this can be especially dangerous in network overload situations where routers are allowed to discard packets to reduce their load.

In Java UDP-sockets are implemented in the class `DatagramSocket`. Concerning the socket there is no difference between the client and the server. There for the same class is used for clients and servers.

2.2.1 The Time-Server

The main part of the time server consists of a while-loop in which the server waits for an incoming message. If it received a message it sends as a result the actual system-time back. The actual system-time is retrieved via the static method of the `System`-class `currentTimeMillis()`. The result of this method is a long integer value. This integer value is given to the constructor of the class `Date`. A `Date`-object can be transformed to a more readable form with its method `toString()` and finally the resulting string is transformed to a byte-stream with the method `getBytes()`.

All data transported over the network using Sockets has to be transformed into a stream of bytes and reassembled on the other side again. This process is also called marshaling.

```
byte[] buffer ;
DatagramSocket server = null;
DatagramPacket packet = null;

server = new DatagramSocket (1777);
while(true){
    buffer = new byte[1024];
    packet = new DatagramPacket (buffer, buffer.length);
    System.out.println("Waiting for time-request...");
    server.receive(packet);
    System.out.println("Connection with: " + packet.\
        getAddress());
    String msg = new String(packet.getData());
    System.out.println(msg);
    System.out.println("Got time-request");
    buffer = (new Date(System.currentTimeMillis()).toString()).\
        getBytes();
    packet = new DatagramPacket(buffer, buffer.length, packet.\
        getAddress(), 1778);
    server.send(packet);
}
```

2.3 Multicast

2.2.2 The Time-Server-Client

The approach for the time-server-client is very simple. The client sends a packet to the server to port 1777. After this it opens a new socket waiting for an answer on port 1778. The client has to create a new empty packet to which the received data can be written. The call to the method `receive()` is blocking. This way the client waits until he gets an answer on port 1778 if there is no answer he will wait forever. In a real system one has to implement some mechanisms to prevent such “dead locks” especially since they are theoretically possible because a lost packet can not be recognized by UDP.

```
byte[] buffer = "time".getBytes();
DatagramSocket client = new DatagramSocket ();

System.out.println("Creating Time-request...");
InetAddress address = InetAddress.getByName("Conan");
DatagramPacket packet = new DatagramPacket (buffer, buffer.\
    length, address, 1777);
client .send(packet);
DatagramSocket clientRcv = new DatagramSocket(1778);
System.out.println("Waiting for response...");
buffer = new byte[1024];
packet = new DatagramPacket(buffer, buffer.length);
clientRcv .receive (packet);
System.out.println("Got response: "+new String(packet.\
    getData()).toString());
client .close ();
```

2.3 Multicast

Multicast is a very interesting feature of the TCP/IP-protocols. It makes it possible to send a single packet to several nodes. Or vice versa one address is occupied by several nodes and packets send to this address are received by all nodes occupying this address.

Multicast is much like radio or TV in the sense that only those who have tuned their receivers (by selecting the appropriate IP-address) receive the information. That is: you hear the channel you are interested in, but not the others.

For the multicast-feature the address range 224.0.0.0 to 239.255.255.255 is reserved. This range of addresses is called class D addresses. Every IP datagram whose destination address starts with “1110” is an IP multicast datagram. The remaining 28 bits of the destination address are reserved for the multicast group to which the datagram is sent to.

3 Remote Method Invocation

Each client which wants to “listen” to the data sent to a specific group has to join this group. In Java this is done with the method `joinGroup()` of the class `MulticastSocket`. To send data to a group a node does not have to be part of it.

A client which listens to a multicast group has to be implemented using the following steps:

1. Get Address of the multicast group
2. Create a socket for multicast
3. Join the multicast group
4. Listen for data

The code example below shows the first three steps. The last step – which is not shown – is to call the method `receive` of the class `MulticastSocket`. This works in an equivalent way as with the other kinds of sockets.

```
byte[] buffer = new byte [1024];
DatagramPacket packet = new DatagramPacket (buffer, buffer.\
length);
InetAddress address = InetAddress.getByName("224.3.4.5");
System.out.println("Wait_for_message...");
MulticastSocket ms = new MulticastSocket (6789);
ms.joinGroup(address);
```

A server or a node which provides data to a multicast group is very easy to implement. Since it is not even necessary to be part of a group a Server does not need to join the group. The code for the server looks very similar to that of the client shown above.

3 Remote Method Invocation

Remote Method Invocation¹ relies on the socket interface. RMI is a higher level interface which provides the possibility of remote method calls. In contrast to CORBA RMI is very simple to implement it does not use a special IDL-language to define its remote interfaces but native Java interfaces. Simplicity is RMI's major advantage.

A major disadvantage is that communication over RMI can only take place between Java objects. It is not possible to mix objects of different languages as it is in CORBA. Both techniques are available on a wide range of platforms

¹RMI

3.1 The Radio Simulator

To implement a real distributable MVC-architecture the radio-simulator is split up into three parts which are capable of communicating with each other with the use of RMI. Below there is a short introduction into those three parts but they are described in more detail in the subsections of section 3.2.

As you can see in figure 1 model view and controller are connected to a network. They communicate with each other via the network using RMI.

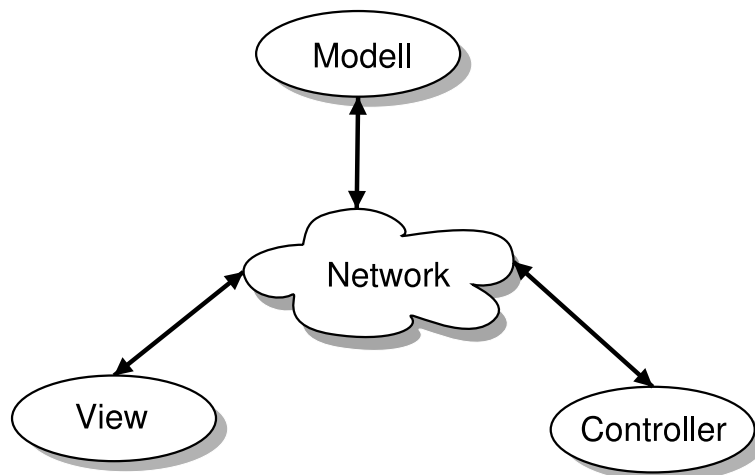


Figure 1: *MVC-architecture distributed over a network*

1. Model
Consists of the “main”-class `Radio` which encapsulates further classes which control the frequencies and the volume (`FrequencyController` and `VolumeController`).
2. View
The view-component is just the display of the radio. The main-class of the view-component is the class `ViewImpl`. This class derives from `UnicastRemoteObject` and implements the interface `View`.
3. Controller
The controller consists of all buttons of the former radio-simulator. Each button is for controlling some functionality of the radio (eg. volume, frequency, ...). The main-class of this component is the class `ControlImpl`.

3.2 Changes in the Design

Until now the radio simulators design was changed several times. The first implementation was a very simple single threaded solution which was not distributable. By pressing one button it was only possible to

3.2 Changes in the Design

invoke one action in the simulator (see [3]).

The next step was to change the radio-simulator to a multithreaded application. In this step the multifunction-buttons were introduced too. By pressing such a button only for a short time it is possible to invoke one action and by pressing it longer than two seconds another action is invoked. Since the introduction of these buttons the amount of needed buttons decreased and there for the GUI changed a little bit. Additionally the “scan”-function was introduced (see [4]).

The third step was to introduce some design-patterns to the radio-simulator and to build a storage system which stores the stations selected by the user. This step was only a minor step because the changes needed in the design were only very small (see [5]).

The last step to make the components of the radio-simulator distributable can be done with very few effort. This step is described more closely in the following three sections. Each section deals with one part of the MVC-architecture.

3.2.1 The Model

For the model only very few changes are needed. The class `Radio` now has to derive from `UnicastRemoteObject` to be able to use RMI. The `main()`-method of the `Radio`-class is very simple as you can see below. It is very important to set a security manager. If you do not set a security manager it is not possible to download stub-classes from other components and there for no communication over RMI can take place.

```
public static void main (String[] argv)
{
    try{
        System.setSecurityManager(new RMISecurityManager());
        new Radio(argv[0]);
    }
    catch(Exception e){
        e.printStackTrace();
    }
}
```

Besides this the `Radio`-class has to derive from `UnicastRemoteObject` and it has to implement an interface derived from `Remote`. This interface is called `RadioInterface` it describes all methods available over RMI. In the constructor of the class `Radio` there was one RMI-specific change needed:

```
Naming.rebind ("rmi:///Radio",this);
```

The rest remains almost the same. The vector which formerly stored references to `Display`-objects now has to store references to objects im-

3.2 Changes in the Design

plementing the interface `View`. All method calls which were directed to a `Display-Object` are now directed to a `View-object` (see section 3.2.2).

3.2.2 The View

In the old non-distributable design view and controller were implemented in one GUI. This GUI used a `BorderLayout` where the display was located in the EAST area of the layout. In the figure below you can see a comparison of the old layout and the new layout of the split up controller and view on the right side of figure 2. The new view just consists of a text-area in which the output of the radio is displayed. The controller which is shown below the view is described in more detail in section 3.2.3.

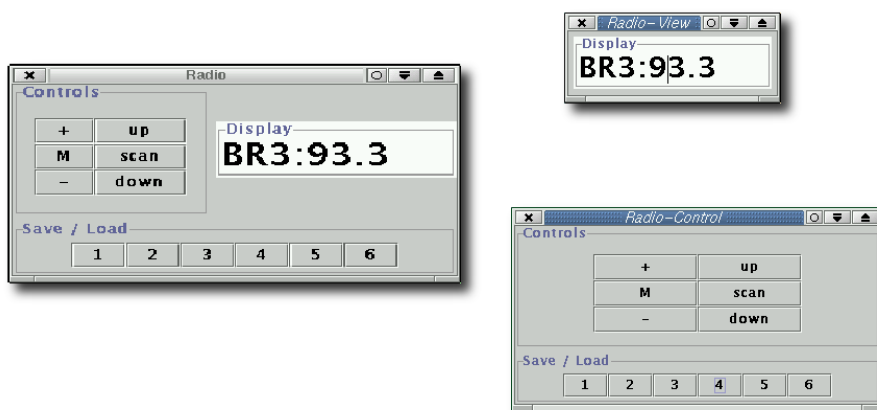


Figure 2: *Old and new layout of the GUI*

I decided that it would be the easiest way to implement a new class for the view which is RMI-capable and just delegates all method-calls to an object of the class `Display` so it was possible to leave the main part of the former view-code untouched. This simple pattern is called facade and is described in more detail in [1]. The facade – which is in our case the class `ViewImpl` – serves as a communication-partner for all other components. In figure 3 you can see the relations between the Java-classes and the classes of the view.

In the old non-distributable design an object of the class `Display` served directly as communication partner for the `Radio`-class (see [3]). For the new design the method calls in the class `Radio` which formerly used an object of the class `Display` had to be replaced by an object of the type `View`. The classes of the view did not have to be changed since the class `ViewImpl` does all the stuff concerning RMI. The view is a “passive” component. It is used by the model (the radio-component) which means that the radio calls method of the view but not vice versa.

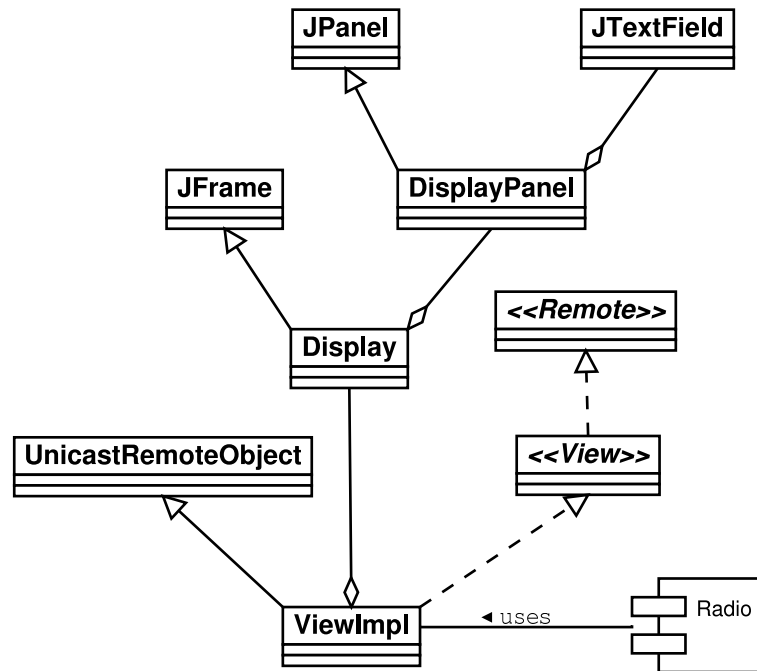


Figure 3: *Class diagramm of the view*

3.2.3 The Controller

As you can see in figure 2 the new controller-GUI is the same as the old GUI just without the text field which was used for the display of the frequency and the volume. Even the layout managers are the same only the text field was removed (see section 3.2.2).

In contrast to the view the controller is a “complete client” which means that it has absolutely no server-functionality – it only calls methods of the model but there is never a component which calls a method of the controller. Because of this fact the controller does not need to implement an interface of the type `Remote` nor does it need to extend the class `UnicastRemoteObject`. During start up the controller just has to search for the model. This is done by the method `lookup()` of the class `Naming`:

```

model = (RadioInterface)Naming.lookup("rmi://" + server + "/" +
    Radio");
  
```

The controller does not “know” anything about the view. There could be a thousand views or there could be none – the controller does not care. It just calls some methods to change the state of the radio and the radio-component itself is responsible for updating the view(s).

4 Conclusions

Socket programming in Java is very simple. A lot of work which is usually done by the programmer is now done by the Java library. The socket-approach provides a very effective facility for data exchange over a network. The multicast feature was completely new for me and there for very interesting. I was surprised how easy it was to implement.

As you can see it was very simple to change the radio-simulator to fully distributable system. Since the radio-simulator already had an appropriate architecture only very few changes were needed. I think by this assignment one can see that Java suits very well for distributed systems. The implementation of sockets is very easy to use and RMI is also kept very simple.

The only thing which causes some problems with RMI is the rmiregistry and the codebase. Since in the beginning it is very confusing if you start the rmiregistry accidently in the same path as your class-files or if you have got one slash too much or less in your codebase-property.

Since this assignment was about two rather large topics it was very hard to limit it to 2500 words. Both topics are capable of filling whole books.

References

- [1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides “Design Patterns”
- [2] Goll, Weiß, Rothländer “Java als erste Programmiersprache”
- [3] Esau “Assignment Object-Oriented Systems Design – A Radio Simulator”
- [4] Esau “Assignment 2 Part 2 Object-Oriented Systems Design – Threads”
- [5] Esau “Assignment 2 Part 1 Object-Oriented Systems Design – Design Patterns and Evolution of Software-Architectures”
- [6] <http://www.ecst.csuchico.edu/~chafey/prog/sockets/sinfo1.html>
“Introduction into socket programming”