Implementation of a Chat System using RMI

Contents

1 Aims and Objectives **3**

2 Technology **4**
2.1 Sockets 4
2.2 RMI 5
2.2.1 Development of RMI applications 6

3 Redesign **7**
3.1 Evaluation 7
3.2 Requirements 7
3.3 Analysis 7
3.4 Design 7
3.4.1 Server 8
3.4.2 Client 9
3.5 Test 9

4 Conclusion **10**
4.1 Outlook 10

1 Aims and Objectives

Distributed systems need methods to communicate. If a system is realized using only Java Remote Method Invocation (RMI) can be used. Aim of this assignment is to become familiar with the basic principles of RMI. To use the techniques a chat system is build. Optimally the chat system that was build for the last assignment can be reused with only small changes.

2 Technology

Big applications have the following requirements:

- consistence of data
- maintainability
- fault tolerance
- load balancing

All those requirements can be achieved by distributing the application over 2 or more hosts. A good solution is to distribute an application over three hosts. One that provides and saves the data, another one, which is called application server, for the logic, and the third host is the respective host of the user that provides an interface. This architecture allows to exchange each of them independent of the others. This architecture is called 3-tiers architecture. An example for this architecture is a system of a bank that provides account management for the user. It consists of a database server, a web server that uses CGI or other methods to provide dynamic content of web pages and a browser that is used by the client. To achieve the four points mentioned above the database can be mirrored to a second server that can replace the first on if it fails. The consistency of the database can be achieved by using transactions. Maintainability is already achieved by using a browser that only displays dynamic generated content so if the program changes this only has to be done on the application server and the load can be balanced by using more than one application server.
The communication over the network has to be fast and secure. Otherwise there is no advantage in distributing the application. To achieve performance the amount of traffic over a network has to be reduced to a minimum or a very fast net has to be used. To achieve security a protocol has to be used that provides a reliable connection over the net. There are several protocols that can be used in Java. The oldest one is to implement a own protocol that uses sockets. Another possibility is to use other higher level protocols like HTTP, RMI, CORBA or COM. The following subsections will evaluate sockets with a proprietary protocol and RMI.

2.1 Sockets

Sockets provide a connection oriented connection if TCP is used. A socket connection can be compared to a bidirectional pipe. After the connection is established data can be transfered byte wise in both directions. The socket itself doesn't know anything about the meaning of the transfered data. So the data has to be encapsulated in a control structure that enables the sender and receiver to interpret the meaning of the data. Methods to compose and decompose the data have to be

written by the programmer. The advantage of self defined protocols is that they don't have much overhead because they are specialized for the appropriate application. But the disadvantage is that it is very difficult and error prone to develop a protocol that is fast and extensible for future use. Another disadvantage is that the programmer has to handle all the possible errors. And the most significant disadvantage is that sockets don't provide method calls. To call methods a protocol is needed, that allows to interpret data and call the appropriate methods on a server. After the method that was called has finished the result has to be transfered back to the caller. This again requires to put the result data in a structure, transfer them over a network and decompose the data. If a method changes the amount of parameters the protocol has to change. This reduces productivity. A great part of the work is spent to implement the protocol.
A possible solution to this disadvantages is to use a already defined high level protocol like RMI. The next subsection will introduce to the principles of RMI.

2.2 RMI

The main features of RMI are the following:

- method calls between different virtual machines
- transparency
- Java specific

RMI provides the call of methods on other virtual machines that are located on the same host like the caller or on machines that are connected over a network. The architecture of RMI can be seen in figure 1. The calls are transparent for the application what means that there is no difference between calling a local method and a remote method. To enable this a "stub class" of the server object has to be compiled using the RMI compiler (rmic). This server stub has to be copied to the client. This stub acts as a proxy for the real server objects. If the client calls the methods of this stub the call is passed over the network to the server. The remote reference layer on the sever side calls the method of a skeleton of the server object that passes the call to the real object. Those skeletons are only needed in version 1.2 and lower of the JDK since newer versions have the reference of the server object already in the remote reference layer. RMI even allows to pass whole objects as parameters of methods to remote hosts transparently for the caller. Those remote objects can be used on the server side like local objects. The server then needs a stub class of the client object. With such an object callbacks to the client are enabled.

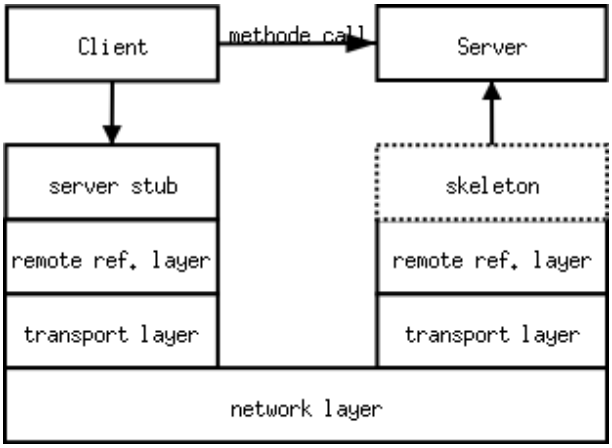

Figure 1: *Architecture of RMI*

2.2.1 Development of RMI applications

For methods that can be used remotely a remote interface has to be defined. The server class has to implement those methods. Further methods can be realized in this class, but they won't be accessible until they are defined in the remote interface. After compiling the server class with `javac` the "stub" class has to be generated using `rmic`[1]. This compiler includes the required communication protocol. The client is developed without regard to RMI. The only thing that has to be considered is that the client can only use methods that are defined in the remote interface of the server. To start the application the `rmiregistry` has to be started on the same machine as the server. The server binds to this registry and registers the objects by name that it wants to provide remotely. Then the client can be started. To access a remote object the client has to do a `lookup` to get a remote reference of the server object. This lookup is done by name again. Normally the name of a service is the same as the class name. An example will be shown in the following section.

[1] RMI Compiler

3 Redesign

Now the chat application that was build for a previous assignment will be evaluated if it can be changed from sockets to RMI without major changes of the design. If the changes are to significant it's better to build a new system. To anticipate the results of the following sections the system will be rebuild from scratch because the approach wasn't build with respect to the possibilities of RMI.

3.1 Evaluation

To decide if the chat system can be extended reasonably to RMI the communication will be looked at. To communicate only `Strings` are send from the client to the servers and the other way without a special meaning. The server distributes incoming `strings` to all clients without looking at the content. Only if the message begins with "disconnect" the the server will remove the client that has sent the message from it's list. As it can be seen here [3] the server is not dependent on the socket connections, it could use any method type of communication. The implementation of the `UserThread` just has to deliver the messages to the `Server` independent of the communication. It would be easy to implement a `RMIUserThread` that could handle the communication with the client using RMI. But then again only `Strings` would travel the network that would have to be interpreted on each side. So the protocol can not be checked at compile time, what is one of the greatest advantages of RMI.
As a result of this evaluation the whole system is rebuild from scratch to have a system that uses the advantages of RMI. The only part that can be reused is the `Admin` class, that provides a web interface for administrative purposes. So the following part will not cover this administrative thread.

3.2 Requirements

The functional requirements haven't changed from those listed in [3]. A non functional requirement has changed: The server of the chat system will no longer be detected using `multicast`. Instead the server and client will be given the name of the host where the `rmiregistry` is running on the command line as a parameter.

3.3 Analysis

The analysis is the same like in [3] and will therefore not be repeated.

3.4 Design

The design of the system is not very complicated. `Server` and `Client` have to be interfaces that implement the `Remote` interface. The imple-

mentations called `ChatServer` and the `ChatClient` have to implement the appropriate interface and they have to extend the `UnicastRemotObject`. The client has to connect to the server using the `rmiregistry` to get a remote reference of the server. It uses the server's `connect` method to connect to pass it's own reference to the server, so the server can send back messages of other clients using the `receive` method of the client.

3.4.1 Server

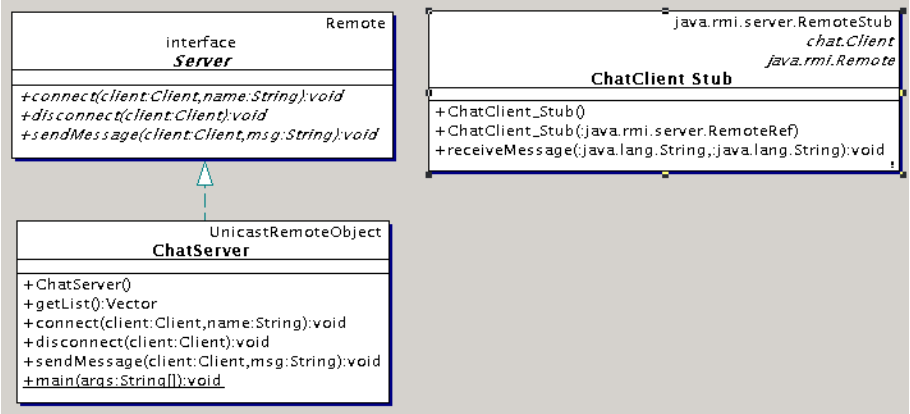

Figure 2: *Class diagram of the server*

The server consists, as can be seen in figure 2 only of two parts: The interface `Server` and the class `ChatServer`. The class `ChatClient_Stub` that is also visible in figure 2 is the class that was generated by `rmic` and has to be copied to the directory where the server is started. Besides the two constructors the method `receiveMessage` of the client can be seen, that is used by the server to send messages to the client. The methods `connect`, `disconnect` and `sendMessage` the server provides have to be declared in the interface `Server`. They are implemented in the class `ChatServer`. The method `connect` has two parameters: a reference of the client and the user name as a `String`, that is used as a key to save the client reference in a `Hashtable`. The other two methods will always check if a client that calls them is already inserted in this table. `disconnect` removes a client from this table and send a message to all the remaining clients that this user has disconnected. `sendMessage` tries to send incoming messages to all clients in the table by calling the clients method `receiveMessage`. If a client is not reachable it is removed from the table and a message is written to `stdout`. The `ChatServer` has no graphical user interface. This is not necessary because it only prints some basic informations to `stdout` and fro further details is provides the web interface that can send the last 10 messages to a web browser. It can be stopped by entering "s" followed by "enter" at the command line.

3.4.2 Client

Like figure 3 shows the client consists of the underlying `ChatClient` that implements the method `receiveMessage` that is defined in the interface `Client`. The `ChatClient` provides methods to register a graphical user

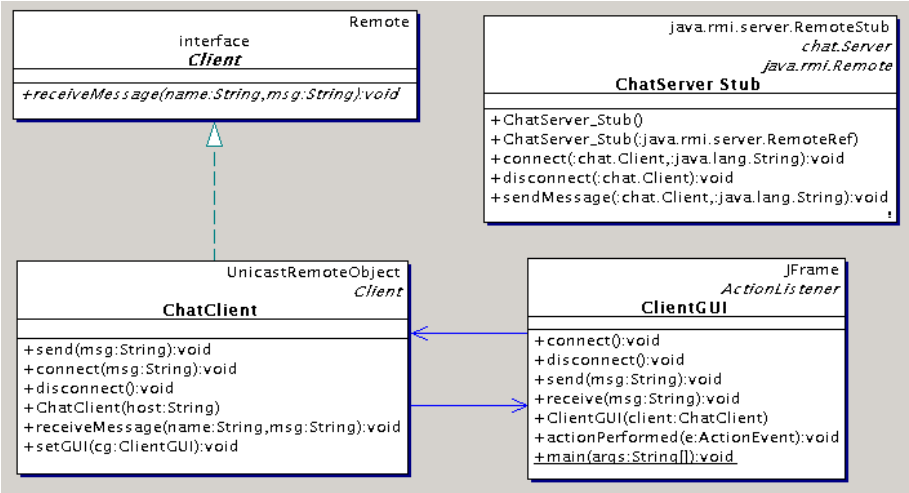

Figure 3: *class diagram of the client*

interface. Here a publisher subscriber pattern could have been implemented more properly, but since the focus of the system is on RMI and there is only one user interface that displays the data and uses the client this has been done very straight forward. The graphical user interface is the same like the one used in [3].

3.5 Test

To test the system a server is started on the local machine (since there is no real network available). Then a script starts as many clients as possible to check if the server is a limited to a particular amount of clients. The limiting factor of the test was the number of threads that could be started on the machine. The limitations of the server will be the maximum number of connections that the operating system allows and the hashtable where the clients are saved will lead to problems, because the access of a hashtable is not very fast, and for each message that is sent to the clients the server has to iterate through the whole table.

4 Conclusion

RMI is a very good way to implement distributed systems if they can be build in pure Java. The programmer doesn't have to care about the distribution during development. So the objects can be distributed dependent on runtime. The protocol of the communication is checked by the Java compiler so the programmer doesn't have to care about this. The disadvantage is the fact that RMI is only available for Java. If server or client can not be build in Java another method has to be used. A possibility can be CORBA or if the the communication has to be fast, sockets have to be used, to reduce overhead, that is inevitable if higher level protocols are used.

4.1 Outlook

The possibilities of the chat system can be enhanced. Normal chat systems like IRC[1] allow the users to create own "rooms" and to change their name without logging out. After the system is running those methods can be added easily. The user interface of the client should be made easier to use. But since the focus was an RMI the graphical user interface was made as easy as possible.

[1] Internet Relay Chat