Implementation of a Chat System using Sockets

Contents

1 Aims and Objectives

Aim of this assignment is to become familiar with th basic principles of networking in Java. This includes TCP, UDP multicast and their usage in Java. To show the possibilities a chat system consisting of server and client is build.

2 Networking in Java

For communication between processes on different machines there has to be a solution, that can easily be 1 by user programs. This solution is called sockets. There are three different possibilities to use sockets: TCP which provides a connection oriented service for transmitting data bidirectional, UDP which provides a connection less service for sending datagrams that don't have to be reliable and multicast, which is based on UDP and allows accessing multiple receivers. They all have one thing in common: since they all use IP which is a connection-less protocol without quality of service, they neither can provide quality of service. So those protocols can't be used for real-time applications. But for a chat application or transfer of data the protocols are very useful. To distinguish different sockets from each other each is assigned a port number, that has to be unique. Some of the port numbers are used for common applications. They are called the well known ports. For example nearly every web server accepts connections on TCP port 80. So the browser tries to connect to this port, if no other port is specified. To exchange data over a connection it is mandatory, that both sides know a protocol. This protocol is dependent on the application and specifies the meaning of the sent data. In the following subsections the three protocols and their usage in Java are described.

2.1 UDP

UDP (user datagram protocol) provides a service for sending data that is less then 64kB. This protocol is connection less. That means that the datagram is just sent and there is no control if the receiver really got the message. If control over reception is needed, the receiver has to acknowledge the arrival on it's own. Protocols like this are used if the more complex protocol TCP is not implemented, or to send data where it doesn't matter if it's not received. For example to view a video stream it doesn't matter if some of the packets are lost because the user won't recognise if a single frame is missing.

2.2 TCP

Unlike UDP, TCP (transport control protocol) is a connection oriented protocol, that is best described by a bidirectional pipe, where on both sides data can be sent end received. The usage is very similar to file usage. First the socket has to be created and opened. After that data can be sent or received from it. To finish using the socket is has to be closed. In Java it is very easy to use sockets. There are two kind of sockets using TCP in Java: `ServerSocket` and `Socket`. A `ServerSocket` can wait for a connection of a `Socket` on a specified port. After accepting the the connection server and client can exchange data in either one or both ways. To read from or write data to a socket the program can get input

and output streams from the socket. Those streams can be buffered, so they only deliver data if a amount of data, for example a whole line of text, is completely received. Using this technique there is no difference between the usage of a file and a socket.

2.3 Multicast

The two protocols described above only provide point to point connection. They do not allow to send the same data to different receivers. For this purpose multicast can bee used. The user only has to specify a group, that is specified by the IP address, that he wants to join. This address has to be in the range from 224.0.0.0 to 239.255.255.255. The receiver can join a multicast group by binding to a socket and waiting for incoming data. Senders just send datagrams to a group using the correct IP address.

3 The chat system

To show the usage of the three protocols described above a chat system is build. It consists of three main parts: the chat server, the chat client and an administrative client.

3.1 Requirements

The system has to meet the following requirements:

- A client should be able to register to the server.
- Functionality to send messages that it receives from one client to all connected clients.
- A possibility for the clients to disconnect.
- A possibility for the administrator to see the last ten messages using a normal browser.

3.2 Analysis

The requirements are leading to the use cases shown in figure 1. The first three use cases are related to the normal user. The last one is for administrative use. Following the use cases are described in detail.

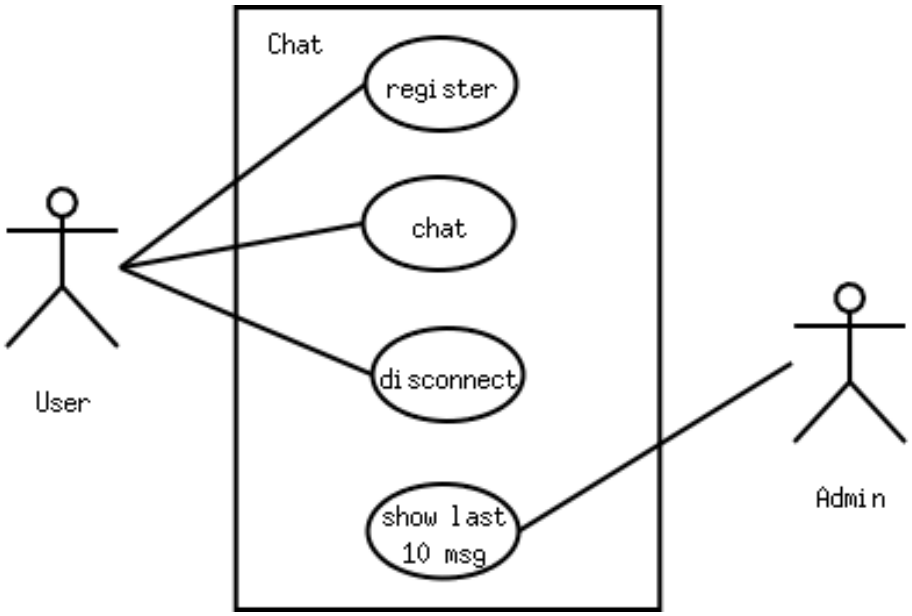

Figure 1: *Use cases for the chat system*

- A user wants to connect to the server. After finding, the server inserts the client in a table, to be able to send messages to each client that has registered. After this the server sends a message to all clients that the user has registered.

- The use case chat contains sending and receiving of messages from the server. A client can send messages to the server. The server sends those messages to all clients. Each client has to wait for incoming messages from the server, to display them to the user.
- The clients can disconnect from the server. The server has to remove them from the list of clients. After this a message is sent to all remaining clients, that the user has left the system.
- To administrate the server a browser is used. The server has to save the last 10 messages to send them to a normal web browser using the http protocol.

Figure 2 shows an activity diagram. A whole life cycle of a client is

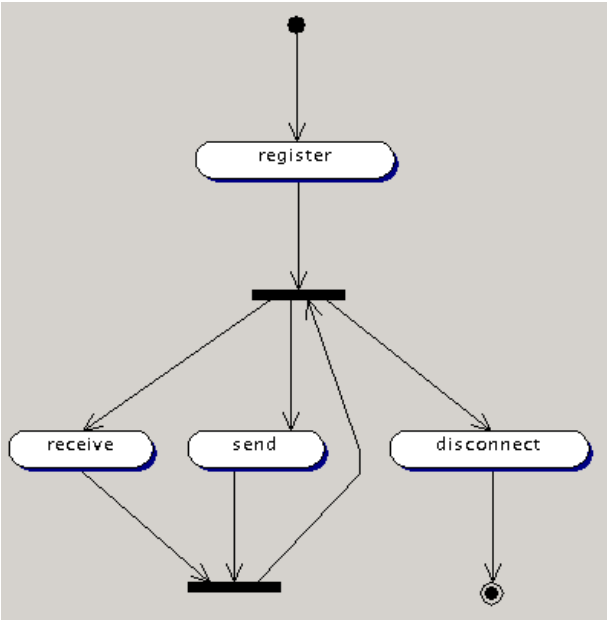

Figure 2: *Activity diagram of a whole client life cycle*

shown. After connecting the client can send and receive messages and then when he has finished he can disconnect.

3.3 Design

The system is divided into two main parts: one for the server and another one for the client. The core of each are classes called `Client` and `Server`. Both have to provide main functionality of a chat system. For the parts dependent on the type of connection interfaces or abstract classes have to be used, so the communication may easily be switched from the socket communication to RMI. In this apprach multicast will be used to find the server. The client emmits a multicast message. An example of a datagramm that is used to connect can be seen in figure 3. The name of the client that is connecting is displayed inverted. In the upper part

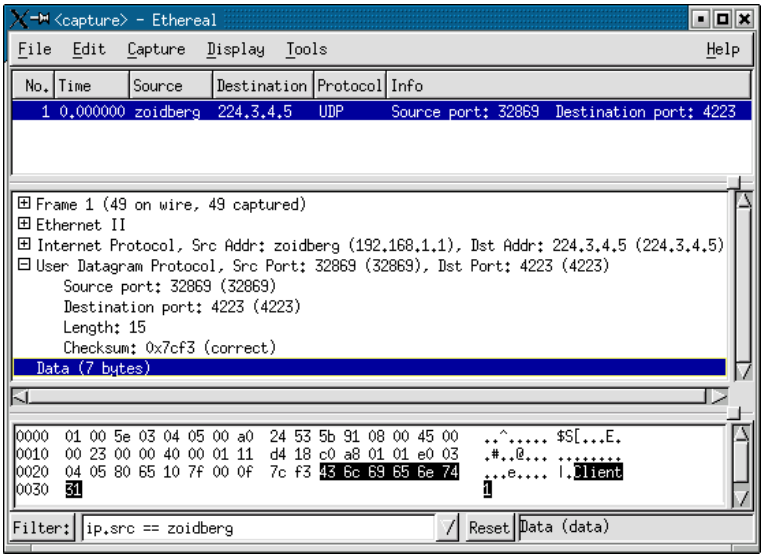

Figure 3: *A multicast packet*

of the figure source and the destination IP addresses can be seen. This also shows that mulitcast is realised by using UDP. The only thing that makes the packet a multicast is the destination IP address. The only thing client and server have to know is the mulit cast group. This could also be realised by connecting to a socket using TCP, but then the client has to know the exact IP address of the server. The only disadvantage is that this system can only be used in an intra net because multicasts are not routed. So it is not possible for the client to connect to a server that is somewhere in the internet. After sending the multicast the client will create a server socket and wait for the server to connect to it. Here the roles of server and client are exchanged, but after the server has connected to the client (he knows the address from the arrived multicast package) both, client and server can get in- and output streams from the sockets and use the streams for sending and receiving. After the connection is established there is no difference between client and server. They both can use the streams equaly. For sending and receiving messages TCP sockets will be used because then server and clients can rely on the connection oriented protocol that is provided by TCP. If UDP or multicast would have been used here the programms would have to make sure that the messages are received by the other side. So a protocoll would have to be implemented, that is already provided by TCP.

3.3.1 The Server Design

To enable the clients to connect while other clients are already using the system, there has to be a part of the server, that is running in parallel, to accept connections. This will have to be implemented as a thread because once a server listens on a specified port, it will not return, until someone connects to this port. To submit the messages to all clients,

references to all have to be stored inside the server. So it is possible to go through a list and send the message to each element of this list. Another approach is to start a thread for each connection. There is an advantage in using this, because if one of the client is stopped without disconnecting from the server the server would have to wait for a timeout. But since the messages have to be passed fast to the receivers for each

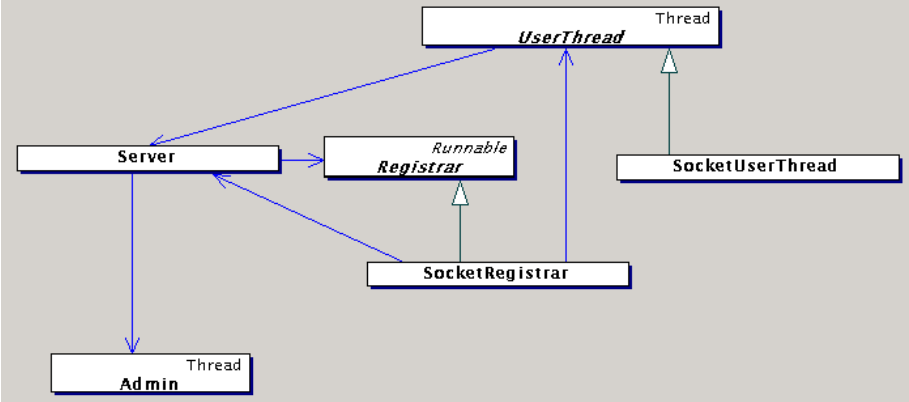

Figure 4: *Class diagram of the server*

client a thread is started. To save memory and the time for creating the threads some clients could be grouped in one thread. Then again if one of them is unproperly disconnected the others will have to wait before they get the message. But for a server where the clients are changing very much this could be useful. The class diagram figure 4 shows the `Server` that uses three types of threads to provide it's functionality. The `Registrar` is used as a sort of portal to the server. The `Clients` have to connect to an object of this class. After this the `Registrar` will create a `UserThread` that is passed to the server to be stored in a list of clients. All this can be done independent of the type of connection. To enable communication using RMI there only has to be implemented the abstract methods in the classes `Registrar` and `UserThead`. The third part of the server is the thread for administrative purposes. This also has to be realized as a parallel thread to ensure communication while the server is administrated.

3.3.2 The Client Design

The client is realized easier compared to the server because it doesn't have to do all the tasks the same time. The main part of the client is in the class `Client`. This class combines model, view and control in one class. That's not a very good design, but since the application is very small an easier way can be chosen. The communication part is realized in the abstract class `ServerConnection` that provides the four methods for communication: register, send, receive and disconnect. A real `ServerConnection` has to implement those methods to meet the requirements of the communication type that is chosen. By using this

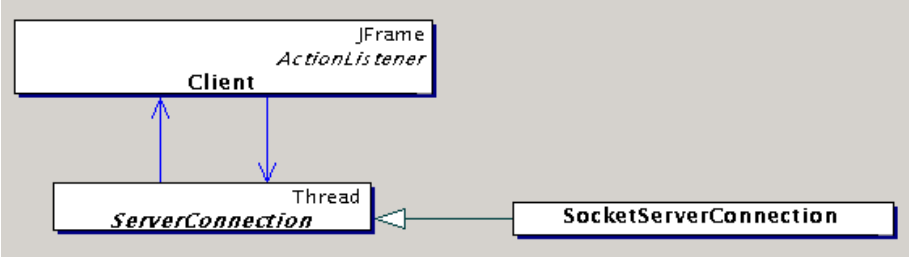

Figure 5: *Class diagram of the client*

abstract class it can be made sure, that the client is independent of the type of communication. So this part can be exchanged independent of the client. In figure 6 a whole sequence of the client connecting to the

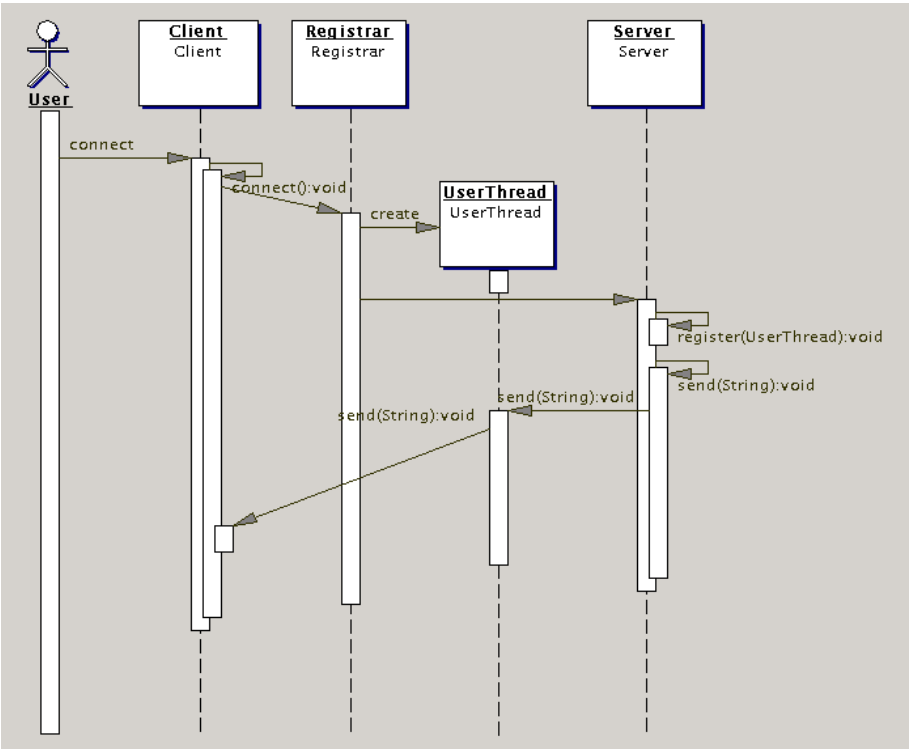

Figure 6: *Sequence diagram of a client connection to the Server*

Server and receiving the reception message can be seen.

3.4 Implementation

Using the results of the previous sections it is no difficult task to implement the chat system.

3.4.1 Client and Server

The implementation of the server is the most time consuming task. Because three threads are running in parallel it has to be made sure that

they don't interfere. It is very important not to mix the functionality of the chat server with the communication to stay independent of the type of communication. For sending and receiving messages socket communication is realized in this assignment. The server doesn't use `ServerSocket` because it is easier if the clients start a `ServerSocket` after sending the multicast message to signal the server that it wants to connect. The server then connects to this `ServerSocket`. After establishing the connection there is no difference between both sides. Either side can send and receive. The client sends a newly generated messages. The server passes this message to all clients of which it has a reference saved in a `Vector`. The clients receive this message and display them using a `JTextArea`.

3.4.2 The Administrator Client

The client for administration doesn't have to be implemented. For this task a normal web browser can be used. The server has to listen on a specified port, to send the last ten messages of the system to this client. Those messages have to be kept up to date by rotating them in an array of `Strings`. For the browser the data has to be sent by using http (hyper text transfer protocol) so it can display the data correctly. The server doesn't use port 80 because this port is already used by a

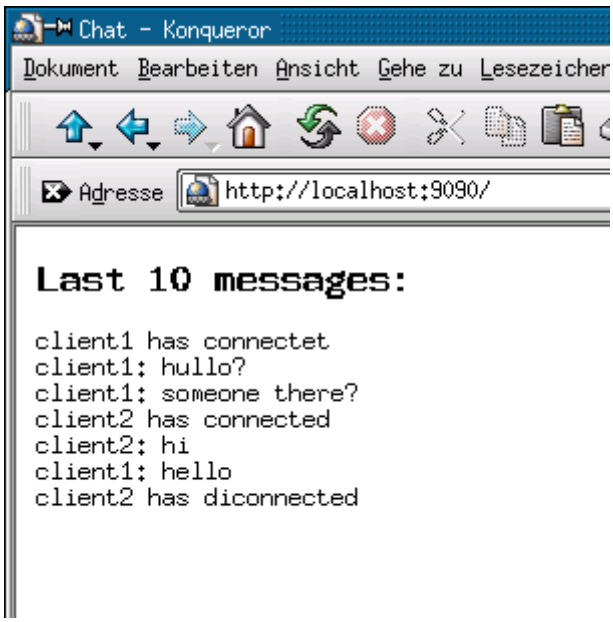

Figure 7: *Output of the administration client*

apache web server running on the same machine. Instead port 9090 is used. To get the last ten messages the address of the server has to be typed in the address field of a web browser. The output is shown in figure 7. If it is used locally the address has to be the following: `http://localhost:9090`.

4 Conclusion

With this application the possibilities of Java using TCP and Multicast have been shown. Java provides an easy to use interface for networking. The system that has been implemented is far from being perfect. The graphical user interface of the client is minimalistic. The server only prints some information to stdout. But since the focus of this assignment has been on networking there was no great attention drawn to other parts. A problem of this solution is that only one client can be run on each host, because the port number the client binds to is the same on all clients. That is the reason why no load test of the system can be performed using only a single machine. A possibility to solve this would be to try to connect to a port first and change the port if needed and then send it with the multicast to the server. The limitation of the server is the maximum number of threads that can be started and the maximum number of connections the system allows. Normally the maximum number of connections on a Linux system is about 4000. What could be changed too is the whole connection procedure. Using multicast doesn't allow to leave the local subnet because multicasts are not routed. Instead the server could just listen to a normal port using TCP. The only disadvantage would be that the client has to know the server's address.