TCP Chat - Technical Report

Server

The server [Server.java] takes as input the port to which it listens. It then creates a socket at the given port. While performing a busy wait incoming requests are accepted and the established connections are created. Finally, the new client is added to the list of sockets to which all messages are broadcasted.

The connection itself takes the according socket and the list of all clients. After establishing input and output streams, it makes use of busy waiting in order to check if there is some incoming data. If there is, the content is read and forwarded to all clients.

Once an error occurs or the connection is interrupted, the server does the clean up an frees the locked socket.

Client

The graphical user interface of the client [Client.java] consists of two parts: the connection establishment and the receiving and writing of messages. The first one allows the client to set the chat server and the port it is listening to. As long as no connection is established, no messages can be sent or received and the corresponding fields are not editable. Furthermore, the user can set his nickname, which is shown beside every sent message, identifying her or him.

The observer classes provide an interface to the classes responsible for the user interface and the connection with the server.

Once the user hits the ‘Connect’ button, the client tries to set up a session. If the socket creation is successful, it too performs busy waiting to check whether new messages have arrived. Additionally, the fields responsible for the connection establishment are set to read only and those used to handle messages are enabled for editing. If new data is available, the messages are passed to the GUI where they are appended to the main text area. Once the connection is lost or interrupted, the used socket is freed for the next client to bind. The same session is also used for sending messages, which are then passed directly to the output stream.

List of requirements:

Your system should allow one to connect multiple remote clients to a single central server.

> Our server implementation allows an arbitrary number of clients, where all of them are fully functional.

When a user enters a text message on his/her client, the message is delivered through the server and displayed by any other client that is currently connected to the server, including the original sending client.

> Every new client is registered in a list with all the other currently connected clients. Once one of them sends a message to the server, it gets broadcasted to everyone of them. The single clients can be identified by their nickname.

Users can join and leave the chat at any time, provided the server is up.

> Our implementation takes care of keeping the whole system in a stable state, allowing the users to join and leave at any moment in time without interfering. Furthermore, there is no connection between the clients and they are not aware of each other. The server alone is in charge of the broadcasting of the messages, adding and removing the clients from its list. This excludes that an error on one client affects the functionality of the others.

Additionally, extensive testing has been performed on any aspect of the system to provide a stable user experience. The results are fully backing up the previously explained concepts. No serious issues have been encountered during the implementation.

UDP alternative:

An implementation using the UDP protocol would certainly work, but for sure not as good as the one using TCP. The first issue is encountered when trying to broadcasting the messages to the other clients. Since UDP is connectionless, the server is not aware of which clients are still online and which have already left. Sure, the client could send a closing message, but this would not work in cases where the user crashes. Furthermore, this would require message parsing and causes general overhead. UDP also does not guarantee that the sent packages reach their destination, which is very inconvenient for a chat application (not loss tolerant). Additionally, no error checking is performed when using UDP. TCP may be slower, but when sending short text messages a delay in the magnitude of milliseconds does not matter.

Therefore, even if UDP is lightweight and in many cases ideal for real time applications, TCP is the right protocol for our chat.