A Client-Server Chat Program

Introduction

The purpose of this assignment is to build a messaging application where users are able to communicate with each other through direct and broadcast messages. The implementation of the application is based on the singe server – multiple client model, which allows a single server to handle all of the requests from potentially multiple clients. What follows is a description of the project implementation, a description of experiments and conclusions made, the features included, data structures used, design choices and instructions on running the program.

Program description

The implementation of the messaging app through the client-server model relies on communication between the server and its respective clients. To enable this communication between the server and a client, the client first attempts to connect to the server and if successful a socket is created on the client side. The server accepts the connection request from the client and creates a new socket to enable communication with the newly connected client while still being able to listen for potential connection requests from other clients. These sockets serve as a sort of pipeline between the server and the client for communicating.

The single server accepts connections and handles requests from multiple clients concurrently which necessitates multi-threading of the server. The main thread accepts new connections and forks threads called ‘workers’ to handle incoming socket connections from clients. Multiple threads are needed because the thread that receives connections is blocked while waiting for incoming connection. The server is thus able to continue servicing other requests while clients are busy connecting or disconnecting to the server.

A connection is established when a local port on the client side is able to establish a connection with the port that the server is running on at the specified address of the server. This connection is handled by one thread on the server side and by two threads on the client side. On the server side the single thread is responsible for handling incoming messages from the client socket and generating a response. On the client side one thread handles incoming data from the server and displaying it to the user in the GUI while one thread handles sending data to the server. The reason that two threads are necessary is because the receiving of data is blocking and thus two threads are needed for a smooth user experience.

After the client has connected to the server, the client has to provide a username and a password matching the username. If the username matches any username currently connected to the server, the login will fail and the user will be asked to provide a different username. Once logged in the user is able to send direct messages to any other connected users, and will receive an error message if an attempt to send a message to a non-existent user is made. The user may also send broadcasting messages, which will be received by every user currently connected except the sender, and will also result in an error message if no other users are connected to the server. Additionally, the user can see the list of currently connected users to establish potential direct message destinations. The user may disconnect from the server at any time and will have to reconnect and login again. Any connection (successful login) or disconnection of other users will be displayed to the user.

All these actions are facilitated by the Server object which provides API's for each task. The worker thread is able to use these methods because the server instance that creates a worker passes itself along in the constructor. This enables the worker to access a list of all workers that have been created by the server. The problem however is that multiple threads could try to access and modify this list and thus whenever accessing this list we have a lock that must be acquired. This ensures thread safe execution of the program. To send messages, broadcast or list users as instructed by the client, the worker thread iterates over the 'workers' ArrayList data structure in the server instance and then sends messages accordingly to the appropriate clients.

Experiments

- Testing maximum client connections (Threading)
  - To see what loads the server can handle we connected over 30 clients to the server. The user experience on the client side seemed not to be affected. We also sent direct messages and broadcasted messages and there was no sign that the number of clients connected to the server had any effect on a client's experience using the service. This was to test the threading capabilities of the server as a new thread has to be created for each new client connection and with 30 clients that means at least 30 threads on the server had to be created. This is more cores than our computer has but the system still worked flawlessly.
  - In conclusion we can say that this system is able to handle a moderate number of clients connecting to the system as would be the case if this application was developed for a small company to use as their method of communication between employees.

- Testing maximum traffic
  - We tested the system by having two modified clients log in and run automated scripts which read a file and broadcasted the contents of the file to the server. This level of traffic would be highly unusual in a normal chat application but we felt it is a good idea to test the limit of the server and see how the system performs under stress. The system performed well and there was no decline in user experience for other clients connected to the server who sent each other messages at a normal rate during the test.
  - In conclusion we can say that the system is able to handle higher levels of traffic than it would realistically experience in the real world under normal circumstances.

- Testing remote hosts
  - In the real world this application would have to run with clients connecting remotely. We ran two simulations to test remote clients connecting to the server. Firstly, we had clients connecting to the server over a LAN by having one machine host the server and two other machines run clients and connect to the server. This worked well and there was no difference in user experience from running the server and clients all on one machine. Secondly, we had clients connect to the server over the internet. We did this by having one computer host the server and then we port forwarded a port on the router that that computer was connected to. We then had two computers from different external networks try connect to the server by connecting the routers external IP address and everything worked swimmingly.
  - In conclusion we can say this this system works flawlessly when the server and clients are all on different networks as would be the case in the real world.

- Testing mass disconnection of clients from server
  - Once we had more than 30 clients connected to the server and the system remained stable (see experiment 1), the robustness of the server was tested further. This was done by disconnecting many users in quick succession and noting the response of the server and the experience of the remaining clients. The server remained stable and could even smoothly accept connection request from clients while the large number of users were logging off.
  - From this experiment it could be concluded that the implementation of the messaging app made efficient use of multi-threading (on the server side) to allow many users to disconnect at the same time without causing the server to crash and would be suitable for a real world situation where many users could disconnect in quick succession.

- Testing smooth user experience (receiving messages and other news while busy typing a message)
  - For this experiment multiple clients were connected and logged in to the server and started sending both direct and broadcast messages to each other. It was noted that a user could seamlessly be busy typing a message while receiving direct messages or broadcast messages from other users. Other updates such as users logging in and disconnecting from the server were also received and displayed to the user while busy typing a message or using any of the other features for that matter. This was enabled by multi-threading different components of the user interface, where a new thread was created to receive messages from the server and display it to the user while the main thread was used to receive the input from the user and send it to the server.
  - From this experiment it could again be concluded that the implementation made efficient use of multi-threading (this time on the client side) to allow users to receive messages as well as other "news" while in the process of typing a message.

Features included

Features included in our solution of the assignment include all of the features mentioned in the project specifications. This includes:

- Username login
- Direct messaging
- Broadcast messaging
- Listing of online users
- Disconnecting of client from server

Features not included

None, all features mentioned in the specification are included.

Extra features included

Connect to server and using application through the command line using telnet works with this application.

File Descriptions

ServerMain:
- This file serves to create an instance of Server and is where the server application is started.

Server:
- This file specifies a server object that uses multithreading to handle multiple client connections. The server also provides an API for various objectives such as broadcasting, sending messages and listing online users. This is all done in a thread safe manner using locks.

Worker:
- The worker is an extension of the Thread class and handles all the individual operations such as logging a user in, and using the server API to accomplish broadcasts, sending message, logging users off etc.

simpleClient:

- Provides an API used to send data from a client to the server as well as receiving data from the server. Includes establishing a connection and logging a user in.

ClientInterface:

- Creates GUI for the user. Contains all methods to receive input information from user and creates a simpleClient. Call functions from simpleClient to communicate with the server and receives information from the server to interact with user.

Issues encountered

We had quite an issue receiving messages on the client side. We were absolutely certain that we were sending messages from the server to the client and that they were being put onto the socket but the client was blocking and not receiving any messages. After many hours we realised that we were not attaching a newline character to the messages send from the server and that the clients call to readLine() on the socket was looking for the newline character before it would return. We were able to resolve this issue by attaching the newline character, "\n", to messages from the server and remained aware of the importance of this character at the end of a message.

Significant data structures

There is a ArrayList object called "workers" for managing all the worker threads that have been created by the server. This data structure is used for in all functionalities of the server such as sending individual messages, broadcasting, listing users etc. This data structure is maintained in the server.java file. This is the only significant data structure used.

Compilation

Navigate to root directory and run 'make'.

Execution

Navigate to the bin directory

- To run the server: 'java ServerMain'
- To run client GUI: 'java ClientInterface'

Libraries

- Java swing
- Java io
- Java net
- Java util

Group members

- Simon Steven 21659583
- Jaco du Toit 22808892