DESIGN DOCUMENT

20205809 - Yanxiu Jin

System architecture

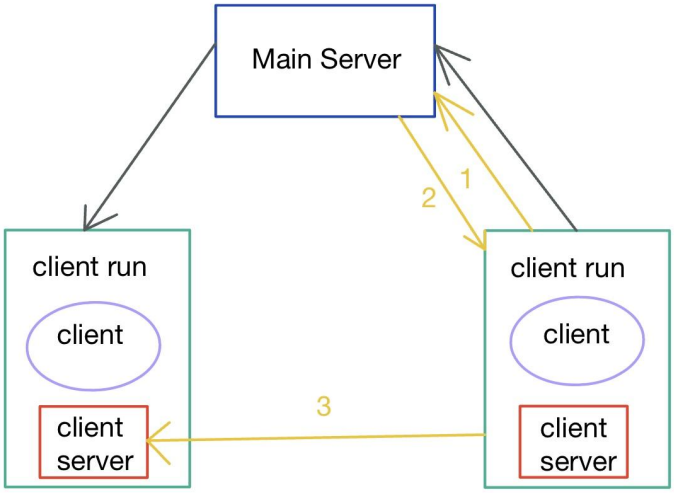

Figure: System Architecture and dynamic connection for broadcast (grey) and message (yellow)

The system consists of two ends: a **main server end** and a **client end**.

The **main server** maintains connection between itself and each client as well as the overall client information such as *clientInServer*, *commands*, name_*port*, which stores all the clients in ***server***, command history by client name and client port number by name, respectively. Every message a client sent is first sent through socket connection of itself to the main server socket. Then a ***ClientHandler*** is created in a thread to manage the client message, whether to send it directly to the receiver's client end (grey line) or tell the receiver which client server to connect to (as it is displayed in the figure above in yellow). If latter takes place, the message is then sent directly to the client server. The GUI is also directly implemented in the *Server* class.

In this system, the **client end** is the ***clientRun*** is a combination of GUI, fundamental properties of a client like its socket connected to the main server, its own server socket and port of client server and username, which in this case is the class *Client*; and structures including connection to the main server through sockets, in which remains two functionalities: *sendMessage* and *listenForMessage*(another thread); and the ***clientServer*** with the help of ***P2PClientHandler***, which manages messages sent to the client server and achieve different tasks on each client end. Intuitively speaking, client server is also a "main server" but under a smaller scale. Since continuous

connection of each peers is not required in this assignment the client server is restricted to the use of receiving private message.

Dynamic Implementation

In order to utilize the system, the user is first required to start the server. The server port in this system is set to 1234. For each client activity, the user should run the *Login* and provide username and port number. A client is generated with these information. Then a *ClientRun* is activated with the given client, which is encapsulated with the *P2PClientView*. The user is then capable of utilizing different functionalities through commands or buttons. There are two types of command requests: "broadcast" type and "message" type.

1. **"broadcast" type:**

   {BROADCAST_CONTENT}
   This request sends the command to the main server and within the client handler, messages with BROADCAST header is detected and the same message with extra information of sender name is sent to all the other clients through the sockets main server is connected to each client, except for the sender. The BROADCAST header still acts as a classification of task within each client.

   {LIST}
   Apart from the header, the only difference for this command is that it only sends the result to the user who sent this command.

   {STATS_ID}
   This is similar to the {LIST}. Since the command history is stored in the server, the only thing to do to gain message is to access the map of commands and send it to the current user.

   s_broadcast()
   A method in the ClientHandler the broadcasts message to all the clients connected to the server.

2. **"message" type:**

   {MESSAGE_ID_CONTENT}
   This request sends the command to the main server and within the client handler,

messages with MESSAGE header is detected and the same message with extra information (receiver's port) is sent back to the client end. After that, a socket, constructed by the port, is created to transfer message with the same header to the serversocket of the receiver's server. Eventually, the P2PClientHandler reads the message, detects the header and display it.

{KICK_ID}
This request is similar to {MESSAGE_ID_CONTENT}, the difference is that in the P2PClientHandler working in the client server, the result of this execution not only prints out certain message, but also removes the client.

It should be noted that whenever a client enters a chat, leaves a chat, get kicked by another client or the server stopped, the announcement is broadcasted. The Exit button closes the window frame and the Help button displays information for the system.

Pros and cons of the System (in the perspective of Decisions)

pros:

a) The server maintains three hash maps *clientInServer*, *commands*, name*_port* that enables faster access to the required information.
b) The p2p connection for the "message" type of command requests highly speeds up the communication process since it is a direct connection.
c) Formatted message exchange for communication between each ends of sockets promotes the efficiency of the system.
d) Both the mouse and keyboard event request is allowed separately to adapt to various circumstances.

cons:
a) The system did not maximize the advantage of a direct p2p as it connects the client server just for sending one message. It would be more efficient if there is a need for continuous private communication or smaller group chats.
b) A new socket is created everytime a "message" type of command is requested. It might be possible of wasting resource and storage especially when there are numerous clients.
c) Part of the user interface is encapsulated inside the back-end, making the code less structured and threatens code readability.
d) The GUI for the Login page has only one input text field where two data is extracted since both the name and the port is required.