1406 Final Project Report

Seth Schill

Description

The goal of this project was to create an OOP oriented centralized chat-room application using a text style user interface (TUI). Additional functionality initial believed to be implemented was a user hierarchy similar to discord's role system, as well as a message indexing database using ADTs. However, due to time restraints the only functionality developed was that of the server and the clients with a crude server handled command system. After cuts were made to other functionalities present in the initial design proposal, the main focus of the project became being able to handle input and output of new and established clients concurrently, without creating downtime for others. In networking applications there are two main ways of doing this which use blocking and non-blocking I/O.

With blocking I/O, the server waits until input is given to it and only after the input is complete can it output anything else to any other clients. In order to provide concurrency in this situation, multiple threads must be created for each client and their respective I/O streams, as well as a main thread which handles clients initially connecting to the server. This implementation is typically fine for a few hundred clients, but for several thousand it doesn't work as for each I/O stream of a client two port addresses must be used. Since there are only 65535 port addresses total (minus ones already reserved or being used by other applications) this is an exhaustive strategy which only holds up for small application uses. For this project, blocking I/O was used along with the threading API for Java since it was quicker to pick up than Java's NIO library.

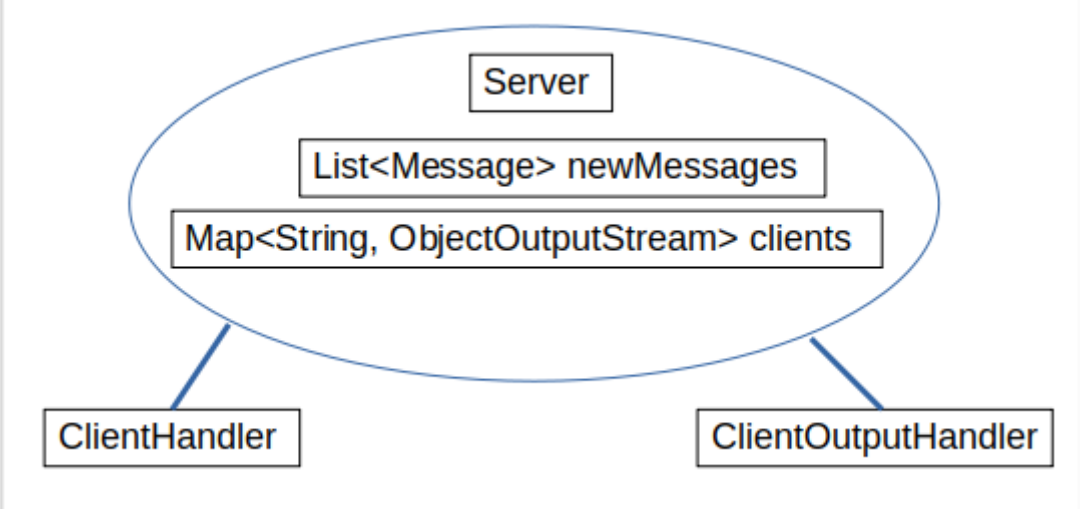
Non-blocking I/O makes use of synchronous communication, allowing for an application to use a single port to send information in and out and the same time. This has many advantages over blocking I/O since, if implemented intelligently, one can handle multiple clients all on the same port.

Implementation

Message:

The Message object is only ever transmitted by the server to reduce client dependence on properties of a message that might be important such as time stamps. A message has three main properties: author, recipient, and content. If the server sees that a message's recipient is defined as null, then it will simply send the message to all of the clients currently connected. However, if a recipient is specified, then the server will send out a private message to them which shows who sent it along with a copy to the sender as well. This functionality is primarily used by the ///msg command, as well as the Server so that other users don't need to be bothered by someone else causing command errors and whatnot. The message class encapsulates all of its data so that none of its attributes can be changed at a later time. In a more advanced system, as initially proposed, involving message indexing, this would be useful since there would be no chance of accidentally overwriting historical records.

Server



Server:
This object acts as the main thread of the server application. It will listen for incoming connections and then hand them off to a ClientHandler thread accordingly. The clients Map and newMessages List are both shared between the server, its ClientOutputHandler, and the many ClientHandlers it may spawn. It should be noted that a Server only contains *one* ClientOutputHandler which is explained next.

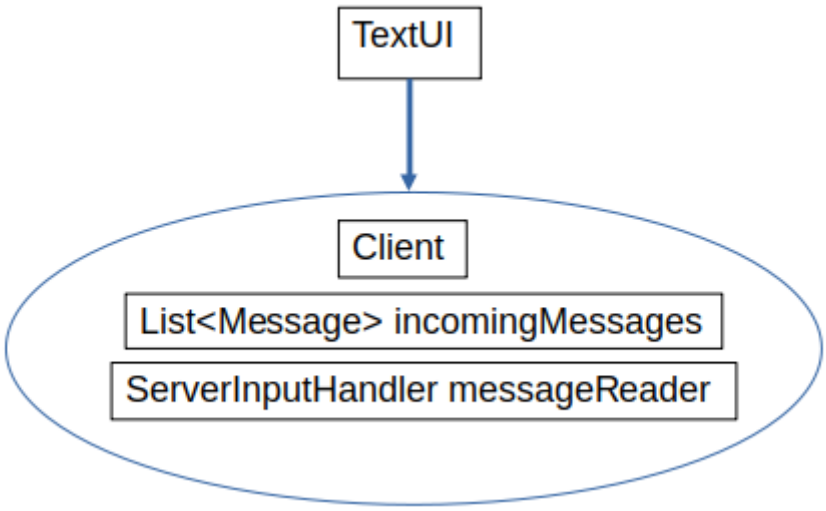
ClientOutputHandler:
This object runs as a separate thread spawned by the server, and its use is to iterate through the newMessages list, and client Map and send out messages to the proper recipients accordingly. The logic behind making this its own thread instead of having it run inside the Server object was to avoid the input blocking present while listening for new clients. It was also better to make it simply iterate over all of the client and messages so that a new output handler would not have to be created for each client. After a message is sent out to all of the clients it needs to, it is permenantly removed from the list to prevent memory build up.

It should be noted that because the list is synchronous and iterated over from the starting index, there is no need for sorting by time stamp. The reason for this is because the server is responsible for time stamping messages and therefore no delay exists as the messages are put in one after another.

ClientHandler:
The first thing done during initilization of this object is the creation of a new ObjectInputStream and ObjectOutStream. After this, a loop is entered which waits for the client to send the server a username which does not already exist in the clients Map. Once that happens, the object assigns itself the clientName String, and adds its ObjectOutputStream to the clients Map, which will then be used by the ClientOutputHandler to send out new messages when they come in.

To make the code more easily manageable, different functionalities were implemented into their own class methods such as sending messages, proccessing commands, handling messages, and handling the login process. The reasoning behind this was so that the model component of the application would be more encapculated. eg.(clients Map, newMessages List). By doing this, the chances of implementing code incorrectly later on in development would be greatly reduced, and if problems did occur then it would be far easier to isolate them based off of the exception handlers present within each segment.

Client



ServerInputHandler:
This object acts as a standalone thread from the Client identical to the ClientHandler object from earlier. As before, all this thread does is read for messages coming in from an ObjectInputStream and then appends them to the incomingMessages List as shown in the diagram above.

Client:
At initialization, this object will attempt to create asocket connection with the server, the appropriate I/O streams, as well as a ServerInputHandler Thread. If this is successful, then the object will set its connected attribute to true, otherwise false. This attribute can be retrieved by a getter function within the object that is later used by the TextUI part of the program to decide what to do next. Because of this possibility, it was made so that the input handling thread would not start automatically, and rather another function was made sot hat it could be started at a later time to prevent the possibility of a runaway thread.

In this application, the Client object acts primarily as the model component of the MVC structure. Its main function is to recieve data from the server and give the TextUI string representations of the messages received in order to further encapsulate the inner workings of the data retrieval.

TextUI:
This object acts as both the controller and view segments of this application. Using the lanterna library, a Text User Interface is created in order to display the data being received by the Client object. This interface is capable of recieving input from the user via text from which it will process accordingly before being handed off to the Client object to send to the server. No sanitization of data is done by the TextUI in order to increase what little security this whole application already has. For example, when entering a username it is possible to put spaces in the name, but the server will automatically turn all spaces into underscores so that commands like ///msg will work properly. If this were to be done by Client or TextUI instead, then anyone could abuse this by sending packets to the server manually themselves in order to get the names they want.

Further Goals

Although obvious by the end of this report, this project suffered some serious cutbacks by the end of completion. Apart from those removed bits of functionality, there are still a few things that could be improved upon OOP-wise. Firstly, the way the TUI and the client are implemented together is fairly troublesome to deal with. Since the TextUI relies heavily on the fact that it's object

has a main loop within in it on which the program runs, it become harder to implement different functionality later on. A better way of doing this would be to completely isolate the TextUI object from the client and have it so that methods from either are called in a separate main loop in order to interact with each other. In other words, removing the client attribute from the TextUI object entirely. The way data like messages is shared between objects is somewhat cumbersome as well, since it requires the reference for each array to be passed to each thread through their initializers. This could be fixed by creating a database system which all objects would be allowed to interact with instead.

External Resources/Libraries:
https://github.com/mabe02/lanterna