Distributed Chat

Introduction

For our project, we created a distributed chat server and client in Java 8. Our message protocol supports the transfer of arbitrary Java types over the network. For the purposes of the project, we implemented standard chat messages, file transfers, and VOIP-like audio streaming. Our implementation is structured into three primary parts: the client, the server, and the protocol used to communicate between them.

Compiling and Running

There are two ways to compile our project. The first will compile all project files and move the class files into a bin directory:

```
make
```

Running the project after compiling can be done like so:

Client	Server
`java -cp bin ChatClient [optional args]`	`java -cp bin ChatServer <port>`

The chat client supports the following optional arguments that can be present (all or nothing):

1. The name of the client
2. The hostname for the server
3. The port that the server is listening on

If these arguments are not supplied, a dialog will be presented to the user that instructs them to fill in the proper information. This information is then stored in a `settings.dat` file that will be automatically read the next time the client is started.

Our project can also be compiled into separate jar files for the client and the server. The commands to do this are:

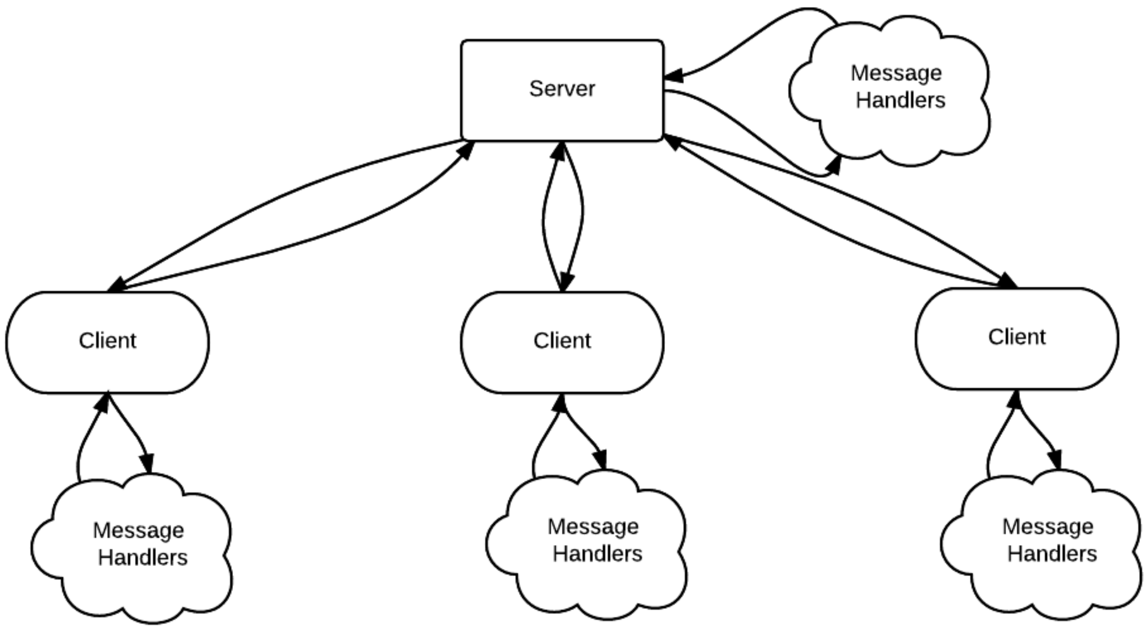
Client	Server
`make client`	`make server`

The resultant jar files can then be run like so:

Client	Server
`java -jar Client.jar [optional args]`	`java -jar Server.jar <port>`

The jar file invocation supports the same optional arguments as the compilation method specified above.

Project Structure



Protocol

The protocol was the segment of our codebase that was shared between both the client and the server. This had the responsibility of describing the structure of messages, the types of messages that could be relayed, and the method by which messages can be handled on either end. There were two primary categories of messages that would be received by the server: communication messages (such as chat messages, audio messages, etc) and command messages (join room, create room, etc). For communication messages, the message would simply be relayed to all clients in the room that the message was sent to, at which point registered client handlers would pick up the task of receiving the message on their end. Commands were handled through the message handlers described in the protocol however.

Message handlers described a single method, `receiveMessage`, which only took the message being received as a parameter. The message handlers were the primary method of implementing messages of different types because each type would have an entirely separate handler that would execute when it was received.

These are the commands that have been implemented in the application:

`/createroom <room name>`	Creates a room that other users can join
`/joinroom <room id>`	Joins the room with the specified identifier
`/leaveroom`	Leaves the room that the command is invoked in
`/listusers`	Lists the users in the room that the command is invoked in
`/listrooms`	Lists all available rooms and their identifiers

Client

The client structure was the model and graphical user interface with which users of our application would interface. This structure handled establishing the connection to the server, handling messages retrieved from the server, and passing messages from the user to the server, while keeping the user informed of all activity in the user interface.

When the user has established a connection, a thread is started that continually listens for messages from the server. Generally, these messages fall into one of two categories: information regarding the activities of other users, or responses to commands sent by the user. Users can interact with the user interface to send audio, file, or text messages, or by issuing the aforementioned commands to the server in the text input box.

Server

The server was a blocking-accept implementation that would offload the handling of individual clients to separate threads that would handle the passing of messages between rooms. Each client handler would listen on the input stream for that client for incoming messages, and then relay those messages on to the rest of the clients in that room. This changed for commands however.

When a command is received by the server, it will offload the handling of that command to a message handler much in the same way that clients handle their messages. This allowed for us to quickly implement new commands by simply defining a method with the right signature and telling the server to call that method when messages of a certain type were received.

The primary client organization structure in the server was in the form of rooms. A room was essentially a collection of clients who have all subscribed to receiving messages from each other. Any client in the server had the ability to create, join, or leave rooms at any time and they would be able to receive messages from anybody else on the server.

Conclusions

Managing a complex protocol between a client/server application was an interesting challenging that we don't often get a chance to tackle throughout the CS curriculum. It provided a good exercise in designing, implementing, and extending a distributed application whose intricacies became more apparent as the project grew. We've discussed what we would have done differently, and there are two major points. The first consists of spending more time developing a message passing protocol. While our system gave us the flexibility to implement all of the features we did, it became more ambiguous and prone to error as the project wore on. The other major point was that, while we feel we did a great job of encapsulating the high level structures of 'Server', 'Client', and 'Protocol', we felt each module deserved a bit more focus in this area. For example, as our project grew, our ChatClient.java file grew to over 400 lines, whereas had we thought a bit more no how to separate the concerns of the model and the graphical user interface, we could have trimmed this file quite a bit. Overall, though, we had more fun with this project than any of our other projects this semester.