Project 4: Simple Chat Server

Release date: Friday, March 23rd, 2018

Due date: Sunday, April 8th, 2018

DO NOT DELAY STARTING THIS PROJECT!

Goals:

- Learn the structure of Client-Server architecture.
- Learn about **Sockets** and **ServerSockets** and how they communicate with each other.
- Understand how Threads are used in the client server architecture.

Prerequisites:

- Understand serializable objects
- Understand dynamic data structures such as an ArrayList
- Understand File IO

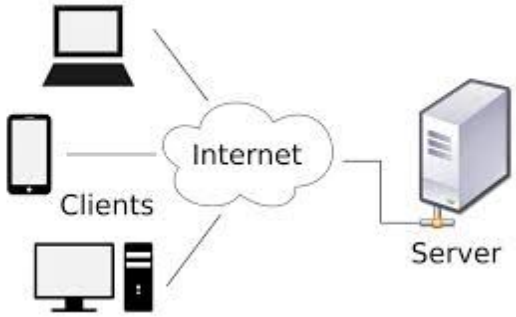
Description:

This project is separated into 6 parts that you have to complete:

Part 0: Running and Understanding the Skeleton Code
Part 1: Making the Connections Persistent
Part 2: Taking Client Input and Sending to Server
Part 3: ChatMessage.java and Tying it Together
Part 4: TicTacToe.java
Part 5: Direct Messaging, Tic Tac Toe, and Listing Users

Introduction to the Client-Server Architecture

Overview: The Client-Server Architecture is an extremely common design for applications these days. The basis of the design is that there is **one** server that processes received data and sends data to **multiple** clients. The server spawns threads for each client that joins. Those threads then wait for the client to send a message to the server.



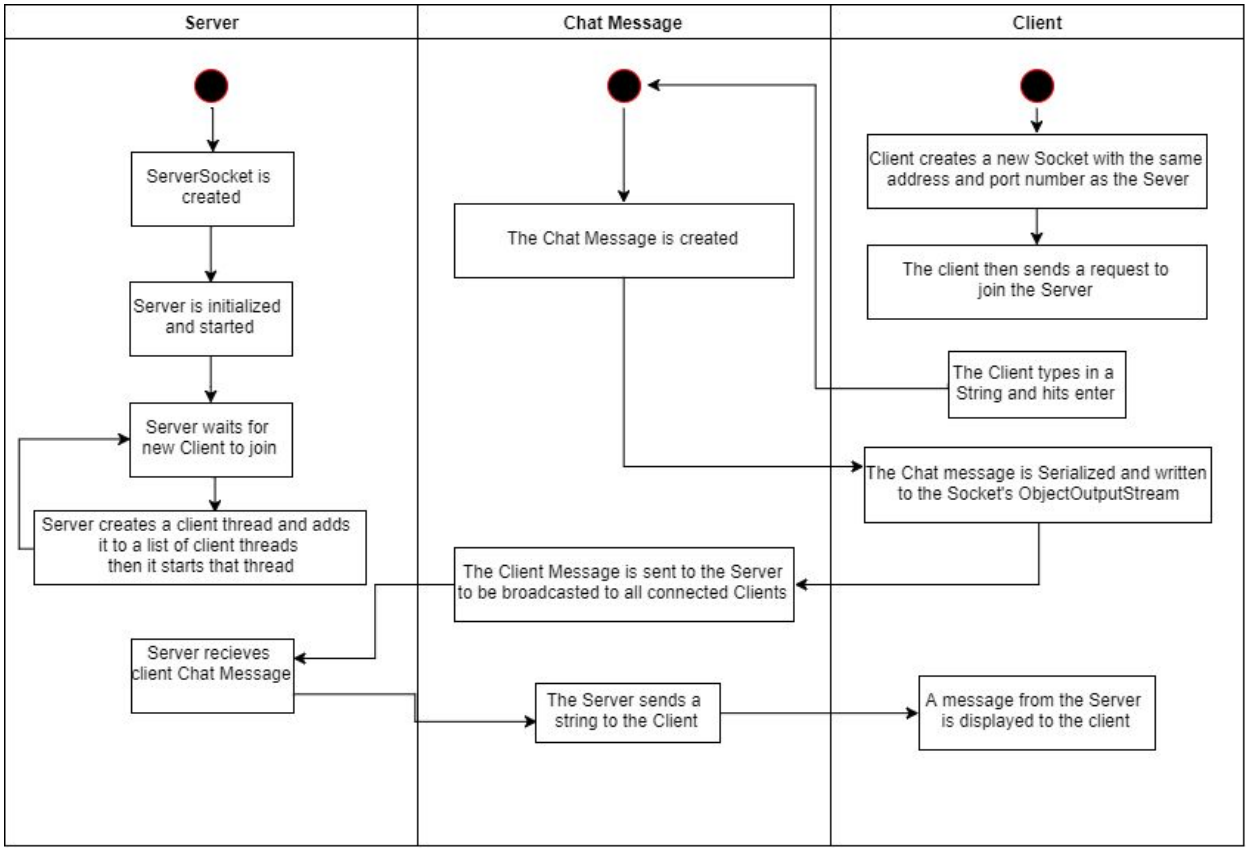
https://en.wikipedia.org/wiki/Client%E2%80%93server_model#/media/File:Client-server-model.svg

In this project, the Client is designed to be independent of the Server, meaning the Server contains no dependencies with the Client. You will be communicating from the Client to the Server through a serializable object. A serializable object is an object that has been converted to a byte stream that is then able to be sent across a connection where it is then deserialized into a copy of the original object. You do not need to implement the serialization.

The server plays a crucial role in the client-server architecture. What the server does is it waits until a client attempts to join a network with identical port numbers. In our case the "network" will be **"localhost"** and the port will be **1500**. "**localhost**" is a special input that specifies that the client Sockets and the server ServerSocket are all hosted on one local machine.

A modern day example of the client-server architecture is a network connection. The router acts as the server, which parses data to be sent into the internet, and your device that is connected to the network is the client. Your device acts independently from other devices on the network. That is unless you are using shared resources. Think how Google Docs works. There are multiple devices writing to one file. From previous labs you know that if these clients aren't threaded properly, the Google Doc would be unreadable from all of the race conditions that are occurring. These race conditions happen because the threads are not synchronized meaning that they will read and write to the file regardless of it corrupting the data.

Activity Diagram of Project 4



Part 0 - Running and Understanding the Skeleton Code

Part 0 overview:

The skeleton code that you have been provided will run (see here for the demo). It does no error handling, user input, message formatting, nor will it persist the connection from server to client. Once the server is started, it will wait for a client to connect and send their username. The server will then print the username and the word "ping". To notify the client that it has received its username, the server then sends back the word "pong". Below is a clearer table of what is happening.

	SERVER	CLIENT
1	Server is started and waits for a client	
2		Client is created and connects to the server and sends a string as its username
3	Server prints username + "ping"	
4	Server sends back "pong" and ends program	
5		Client receives "pong" and prints it and ends the program

Questions to ponder when reading the Skeleton code:

- Why do we need to start the server before starting the client?
- How are the messages being sent from the server to the client and vice versa?

Part 0 notes:

What are ArrayLists:

An arraylist is a dynamic data structure. Dynamic means that it adapts as data is added to it or removed from it.

To initialize an ArrayList you do the following:

```
ArrayList<Type> dynamic_array = new ArrayList<>();
```

The word Type is a java object that the list contains. So if the Type was String , you would have an ArrayList of strings.

To add to an Arraylist use the **add** method. To remove, use the **remove** method. To get the size use the **size** method. To get an element use the **get** method.

For further explanations and details see the javadoc for ArrayList.

Part 1 - Making the Connections Persistent

Part 1 overview:

Right now when you run the skeleton code you will notice that the client sends a message to the server and then both the client and the server close their connections to each other. Your goal is to make those connections persist so multiple messages can be sent without restarting the server and the client.

Part 1 details:

In order to for the server to persist forever, you need to make the program run forever.
To do this you will need to implement an infinite loop that indefinitely accepts clients connections.

Example:

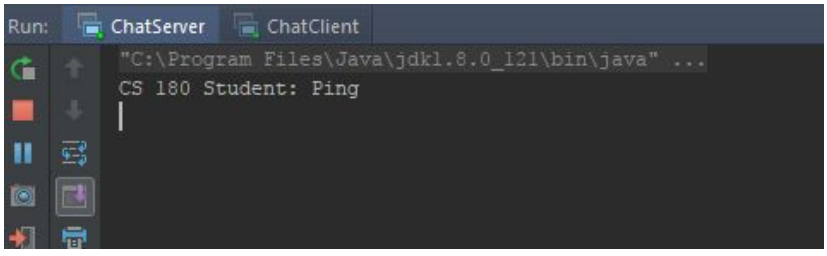
```
while(true) {
        Socket socket = serverSocket.accept();
// server waits until a client opens a Socket with the same address and port number
}
```

Once this is implemented, the server will run indefinitely. But the client still doesn't persist after the "pong" message is sent from the server to the client. Once again you will need to implement an infinite loop that listens for the server to broadcast messages to the clients.
In the **ListenFromServer** class's **run** method, you will need to put the infinite loop there. This method will continue to run

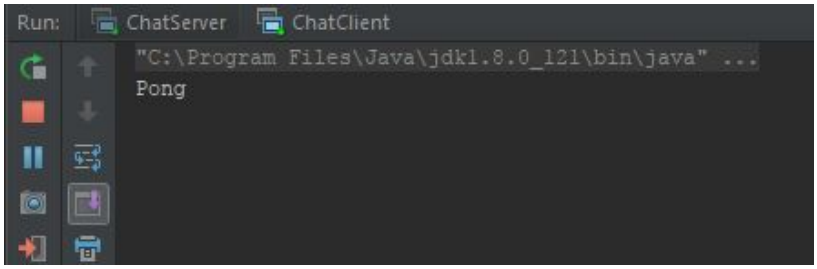
Part 1 demonstration:

Once implemented correctly, your server and client should behave like below. Note that the program does not end after printing these messages.

Server



Client



Part 2 - Taking Client Input and Sending to Server

Part 2 overview:

In order for the chat server to be useful, there needs to be chat functionality. In your main method for the **ChatClient** class you need to be able to take in parameters like username, server address, and port number.

Example: in terminal (or IntelliJ configuration)

java ChatClient username portNumber serverAddress

The example above assumes:

- **username** is an alphanumeric string with no spaces.
- **portNumber** is a numeric value.
- **serverAddress** is the ip address of a server or can be “localhost” if unspecified.

Below are the following formats of parameters that need to be handled:

java ChatClient [username] [portNumber] [serverAddress]
java ChatClient [username] [portNumber]
java ChatClient [username]

For the server, the parameters passed should follow the format below:

java ChatServer [portNumber]
java ChatServer

Note:

Fill in the missing parameters with “1500” if portNumber is not specified, and “localhost” if serverAddress is absent.

Now that parameters are dealt with, we will move onto the chat functionality.

Implementing the chat functionality will be done in the **main** method of the **ChatClient** class. There you will need to:

1. Implement a way to read in user input from terminal after the **ChatClient** object has been started.
2. Parse the user input to determine the type of message.
3. Send the message with the **sendMessage(ChatMessage msg)** method (more detail in **part 3**)
4. Repeat from step 3.

Now you have chat functionality. One more thing though, you need to also implement a way for users to **logout** from the chat server.
To handle this you need to:

1. Check if the user inputs "**/logout**" (case insensitive).
2. Close the client's ObjectInputStream, ObjectOutputStream, and the Socket.

The server will handle the logout in the next part.

Part 3 - ChatMessage.java and Tying it Together

Part 3 overview:
In this part you will define the basic attributes of the **ChatMessage** class. Right now, all you should see is **private static final long *serialVersionUID* = 6898543889087L;**

This is how you will serialize your messages, effectively keeping the client separate from the server. In basic terms, serialization is the process of converting an object with data into bytecode where it is then transmitted across a network connection and reconstructed as a copy on the server's side. You will **not** have to implement the serialization, but you will create the rest of the **ChatMessage**.

Constructor
The constructor will need to take in an integer and two Strings.

The integer determines the type of message that will be sent. The types of messages that can be sent are **0** which is a general message (nothing special), **1** which is a **logout** message, **2** which is a direct message between users, **3** which is a request for the list of clients, and **4** is a request

to start a TicTacToe game with another user. The **logout** message, when reconstructed in the server, will let the server know who logged out.

The first String is the message that will be sent from the client to the server, and the second String is the recipient of the message. If the message is not a direct message or a game command, the recipient field can just be an empty String.

Methods

You will need to add **getters** to get the **type**, the **message**, and the **recipient** of a **ChatMessage** object.

Serverside

Now that the client has the option to logout and send messages we need to handle that in the server. To do that you will implement the following methods:

- **private void broadcast(String message)**
    - The **broadcast** method will be shared across all clients so this method will need to use **concurrency**. This is up to you how to implement this. The method will then print the message to terminal of every client. This will be done by iterating through the client list and writing the message using the **writeMessage(String msg)** method. You will also need to print the message to the server's terminal with a simple print statement.
    - When broadcasting the message to the server and the clients, add the date and time when the message was sent. Do this with SimpleDateFormat using the format "HH:mm:ss".
- **private boolean writeMessage(String msg)**
    - This method will need to be implemented in the **ClientThread** class inside of the **ChatServer** class. **writeMessage** will return false if the socket is not connected and true otherwise. Before returning true, make sure you actually write the message to the **ClientThread**'s ObjectOutputStream using the writeObject method from the ObjectOutputStream class.
- **private void remove(int id)**
    - The **remove** method will take an integer as input that is used to determine which client to remove from the **clients** ArrayList. This method will also be shared across every client thread, so once again you will handle the **concurrency** here.
- **public void run()**
    - The **run** method is partly implemented already and is located in the **ClientThread** class. What you need to do is to handle the **ChatMessage cm** by checking which type **cm** is and handling it appropriately based on the **type**. You will then **broadcast** the message to the server in the format: username + ": " + message

- **private void close()**
    - This method does the exact same as logging out in the **ChatClient** class, but for the socket connection from the server to this client.

Note:
At this point your server is almost done. Now you need to do some error handling.
Think of ways your server can break.
Think along the lines of "What happens when I start a client before starting the server?"
Also note that **usernames are case insensitive and unique** so there cannot be two names that are the same.

Part 4 - TicTacToeGame.java

Part 4 overview:
Now that you have your server and chat up and running, let's add something fun for users to do while connected! We are going to be implementing a small Tic Tac Toe engine into the server.

Tic Tac Toe is a fairly simple game. It takes place on a 3x3 board of spaces where the two players alternate placing their respective character, X or O, into an open space. The goal of the game is to get three Xs or three Os in a row while, at the same time, preventing your opponent from achieving the same thing.

Sample Tic Tac Toe Boards

```
 0 | 1 | 2              X | O | X
-----------             -----------
 3 | 4 | 5              X | O | O
-----------             -----------
 6 | 7 | 8                | X |
```

With the game in mind, you will be implementing a way for users to play matches of Tic Tac Toe with each other through the Client-Server connection.
Users will play with the /ttt command as follows:

To start a game against a player:	`/ttt <opponent>`	`/ttt Anna`
To make a move in a game:	`/ttt <opponent> <space>`	`/ttt Anna 3`
To view a current game's board:	`/ttt <opponent>`	`/ttt Anna`

Some notes:
- The player that starts the match with /ttt will be player X. The <opponent> is player O.
- Players cannot start matches with themselves.

- Assume a new game cannot start with the same player until the previous game finishes.
- Players may participate in as many matches as there are current players simultaneously.
- Players should not be able to play out of turn (like twice in a row!)
- A message should be outputted to both players when a game of Tic Tac Toe is started or completed.
- The board should be shown to both players after one of them makes a move.
- Make use of return values from your methods. For example, return a negative number if something fails. You can even return different negative numbers for different failure cases (this is useful in testing!)

Some other things to think about:

- How should the server handle making a move against a disconnected player?
- What if that disconnected player reconnects?
- Where should the games be stored? Globally by the Server? In the Client threads?
    - (Pick a method that makes sense to you!)
- Where is the game input processed? Server side or client side?
- What controls creating and ending the games of Tic Tac Toe? If TacTacToeGame is the game itself? (The server probably needs to process this!)

You may wish to look at the provided **directMessage** code in the skeleton to get some ideas on how you want to handle the Tic Tac Toe games and user input for them. Direct Messaging works similar to the server's broadcast function but only sends its output to a single user.

Suggested methods in TicTacToeGame.java:

Name	Return Type	Parameters
`TicTacToeGame()`	-	`String otherPlayer, boolean isX`
`takeTurn()`	`int`	`int index`
`getWinner()`	`char`	-
`isTied()`	`int`	-
`getSpace()`	`double`	`int index`
`toString()`	`String`	-

Feel free to implement additional methods if you think they may be helpful to you.
Server-side, you probably want some way to *start a game, process a turn,* and *end a game*.

Part 5 - Listing Users

Part 5 overview:

Listing Users

Your server should also have the ability to list the users that are currently connected to the chat server. The format for this command will be:

/list

This command will print the names of the users who are connected to the server **excluding** the user who typed the **/list** command.

Demonstration:

For a demonstration of expected server-client behavior, please see the video demo provided here.

Turning in Your Work

Submission

Submit the following to Vocareum prior to the due date:

- **ChatServer.java**
- **ChatClient.java**
- **ChatMessage.java**
- **TicTacToeGame.java**

Team Review

After the assignment due date, you will be requested to provide us with feedback for how you and your peer worked as a team. More details for this will be given nearer the due date.

Grading

Grading will occur in lab the week after the due date and will be purely manual. At least one of your teammates must be present for questions regarding the program (there will still be a lab assignment, as usual).

The grading rubric is on the following page.

Rubric:

Task	Points
A user can join the server and is greeted with a welcome message from the server.	7
User can send a chat message and all other users will receive it.	10
The user can send a chat message and the server receives it.	7
The user receives an error attempting to join if the server is not running.	3
Multiple clients can join the server at one time.	10
The client and server persist after one command. (i.e. they run forever)	10
Users can logout using /logout. The server then broadcasts that the user has logged out. (The client list is updated)	7
The server should not disconnect unless the server is stopped in terminal (Ctrl-C)	3
Two clients cannot join with identical usernames.	7
Direct messaging works with two separate clients. Direct messaging works with two or more separate clients. (a client cannot direct message themselves)	7
/list command list the users in the current chat server excluding the user who typed the command. /list command lists the users in the current chat server excluding the client who typed the command.	8
Users can play Tic Tac Toe with each other.	7
Tic Tac Toe games won't start if they are already in progress, and will end properly.	3
The format of output is similar to the demonstration. (including the simple date format)	3
There should be no race conditions when sending messages to the server or disconnecting users.	3
Team Review	5