COMP90015_ChatRoom

Kunliang Wu - 684226 - Information Technology

This report is written using Markdown.

Inroduction

This project uses some basic features of JDK to implement a Simple Multi-User ChatRoom.

- **Github Page** https://github.com/Unibrighter/COMP90015_ChatRoom
- **Third party library** args4j-2.0.21 json_simple-1.1 log4j-1.2.17

Some Challenges for the project

As a small distributed system, it has some basic challenges that almost every distributed system would face. - **Concurrency** We have to face multiple connections from time to time in this chatroom. If there are several clients try to use the broadcast channel, and all of them write their try to write something to the outputstream to the chat room, collision will surely happen and what we get would be a shattered message messed with each other. We have to make some restrictions and stop this kind of situation, a good solution is to use a lock for the broadcast channel in the chatroom.

```
Furthermore, another idea is to optimize the protocols to get rid of the server,
the clients can have a list of other clients who are online, at the cost of large
memory. Or we can do a better job using mixed method - **So that the client-list is
kept at server side, but the messages are transmitted among clients without the
server's knowing.** That's how BitTorrent Protocol functions.
```

- **Data Integrity** Think about the situation when two clients apply to create a new chat room at almost the same time. If there is no check about data integrity, both of the threads dealing with each users would consider they have successfully created a new room, then who will be the **real owner**?Here we need to make sure the checking logic has to be atomic, in other words, the code section has to be marked by ***synchronized***. Another way to solve this problem would be placing the owners of the room in a list in ChatServer.java, and mark it as ***volatile***.
- **Data Consistency** If a user named Alice kick another user named Bob, can Bob re-enter this chatroom just by changing his identity(Id)? If the owner of the chatroom has left, is it still the same case? These problems have to be carefully taken care of by giving a well architecture of the program.This requires us to wrap the client as a Single Class, and the black list should no longer take identity as the valid way to identify a client. I have put some works into this problems and found that a more efficient way to solve the problem is to assign a hidden serial number to each client when there is a new socket connection established.
- **Independent failures and handling** I will discuss about it in the following section.

Other problems I encountered in the project and my ideas and solutions

1. When interpreting some responses from the server, it's quite hard to tell the information responding to our former request we just sent out from those information we receive from the server passively.For those request like creating or deleting a room, we need to checkout the room list before and after we send out the request. This is both annoying and time-wasting. If I am to build a better version of the protocol, I will simply add a new boolean attribute called "Success". Then we can understand if this is a certain response for our request just now, if we see there is no such a attribute in the coming json, we can conclude that it is just another passively message which is broadcasted over the whole channel.
2. As for the server program architecture and way to manage the clients cluster, I have two totally different ideas.
   - We consider each chatroom as an instance of a class named ***ChatRoomManager.java*** . In this way, we can get access to the entire chatting group within a room by calling the function of ChatRoomManager.java
   - We consider each chatroom as a tag(or an attribute, to be precisely). Every time we are sending a message, we do the adjustment according to the attribute in ClientWrap. In this way, chatroom tag is to a client what a mask to an IPv4 address.

   The first one fucuses more on concurrency and the management of one single chatroom, while the second one, is more light-weighted and can adapt to possible future extension.Personally I picked the first one, I can set several synchronized methods in ***ChatRoomManager.java*** class, like "adding/removing a room member". Also we can easily better the performance of server end by set up a thread pool inside the class and let JVM do the rest.
3. In the workshop, the tutor introduced us way of managing the input and output stream ,using some pre-defined methods like **wait()** and **notify()**. This is not always the best the approach when dealing with two threads specializing in input and output separately.We can using only one thread to deal with the inputstream and react according to the json content, since the BufferedReader will keep on reading the input until it reaches the end (end of file or stream or source etc). In this case, the 'end' is the closing of the socket. So as long as the Socket connection is open, your loop will run, and the BufferedReader will just wait for more input, looping each time a '\n' is reached.Also, there is no need for us to add a '\n' after the message string, because BufferedReader already provides us a method called 'println'.This is another main reason why I used ChatRoomManager as a class to manage the clients within a chat room. I consider they all belong to one channel, this way I don't need to worry about extracting information from all different clients using a loop, since a loop will get stucked atinputStream.readLine();
4. Some may argue it's a waste of resources and time but still, I did the failure and validation check at both the client side and server side. There are two reasons why I did it.
   - For Server's side, the general philosophy in interaction programming is “never trust the input.” It's a consideration of both security and logic. Some malicious input may reach your server via others Client Terminals which is not designed and implemented by you.
   - For Client's side, we should try our best to filter out those illegitimate request to less the pressure for server.

Some ideas about a possible multi-server architecture

Approaches to a possible multi-server architecture

There are two main categories that would work : 1. we can try to solve the communication among different servers by setting up a ***Server Of Servers*** to make the servers cluster function as a single unit, which is opaque to the clients. 2. The servers communicate with each others and communicate without a centralized node. They sort things out for themselves.

Both of them requires to set some extra protocols and we also need a distributer to assign each server with the most appropriate task. For example, we can use ***ping*** to give us some general evaluation about the shortest serving time before we determine which server is the best option.

Centralized Mode

This mode performs well referring to data insistency and concurrency.

We can set up a look up table in the central node, and flush for each period. We can use a unique number to represent the connection to avoid the problems caused by name changing protocols. The central node can keep track on which clients are connected to which servers, who the members are for a given room id,if there is a new room has been created as well as other similar global data and statics.

The server can upgrade the status info to the central node using an array taking a json form. But the real message exchange ("type":"message") are still done by negotiating with other severs directly.

But this mode has a huge defect - once the central node is down, the whole system crushes.

Peer Mode

In my opinion , I think this mode is more common and suitable if the data weights much more heavier than meta-data. In other words, most of the data transmission is the contents' load, like big file over the network.

So that's why I think Centralized Mode is better for this project.

For a multi-server architecture which can scale according to the requirements, we need to minimize the amount of data exchange. If the delay is not so important and we can tolerate it to some degree, I think we can come up with a protocol that resemble the algorithm like a minimum spanning tree.

Then the duty of each individual server is informing its neighbor about the information it gets from the its clients cluster. Some discover path algorithms can help condition the server cluster to deal with some changes and individual breakdowns.