1549 Advanced Programming

Alexander Askew, Romans Porubovs, Kamelia Ikonomova & Younies Bayoumy

Abstract

In this report we will be discussing different aspects of making a peer-to-peer messaging service with more advanced programming concepts. We will discuss how we designed and implemented the specified features as well as the testing we did on the software we had written.

Introduction

As a group we are familiar with the basics of programming. We knew about object orientated programming as well as using functions but there programming is a vast subject and there is always new stuff to learn. Advanced programming is taking what we know and teaching new concepts such as modularity, and with these concepts we have made a peer-to-peer messaging service. As a team, we sat down and talked to each other over text and phone calls to work together to produce our messaging service. We decided to use python over java as we came to the unanimous agreement that python was the language, we were all most comfortable in. We created a server and a client that uses python sockets open a port and listens to each other. In this report, we will go through and all of us will explain; how we went about designing the software and its features. Speaking of features, we will always explain in depth about how we implemented them and justify why we chose some features over others or why we cut some from the final product. Once the design and implementation is discussed thoroughly, we will then focus on testing our software and analysing it. There will also be a discussion on whether our software/code meets the requirements set by the specification, such as modularity.

Design and Implementation

For creating our software, as discussed in the introduction to this report, our group decided to write in the programming language python instead of java. We came to this agreement as we all agreed we were most comfortable with python's structure and syntax and thought this would be our best language for learning new concepts. One of us preferred using the native IDE that comes with python, however when it came to the rest of us, our contributions were made in PyCharm as it's an intuitive environment that has colour coding which allows us to look through and troubleshoot code much easier. Our software consists of two parts, the client and the server. This is in the form of a star topology meaning that you have multiple clients, that will be the users talking to each other, all connecting to a central server. The advantage of doing this is that clients should be able to come and go without disrupting the connections of the other clients. Of course, the downside is that if the server comes down then all the connected devices are not able to communicate with one another privately. There were some specified requirements for this project. Firstly, each member must have a unique ID. Secondly, each client opens a port in which it listens from. It also should have its IP available. Thirdly, the server itself needs to have a designated port and an IP address. Next, this is where some of the most in-depth code comes in, the first client to connect becomes the coordinator of the room and can see all the subsequent connections' IPs, ports, and IDs. Essentially the rest of the specification is the roles that the co-ordinator plays, and clients should be able to receive and send messages to the server. Before we talk about the code and designs, we will quickly discuss the look and design of the program. The specification states that the program can either be a command line interface (CLI) or have a graphical user interface (GUI). When discussing the process of completing this project, it made sense to worry about a GUI last as it involved putting graphical elements on top of the backend code. In the end we decided against making a GUI with the justification that it offers very little in the way of functionality and just provides the user with something unique or interesting to look at. Next, we should talk about one of the new concepts that advanced programming has taught us, modularity. Modular code is kind of what it says on the tin, the code should not be blocks of hardcoded functions that only work in the context of the program. Some of our early prototypes of the code relied heavily on hardcoding everything to the program. This meant long blocks of code that constantly referred to itself. This mean that the code could only be used for the one job we were using it for now. As we went on and learnt the new libraries we used, it became easier to make our code more modular. It isn't so modular where you could straight up take out of our program and utilise in another program but our use of many functions that called to in other functions would, in theory, allow us to use them in other programs with some changes to adapt to each program. We managed to implement most, if not all, of the required features. First, we have successfully created a star topology and got a client - server connection working. We achieve this by using sockets, which are used for low level networking but work perfectly for a few connections sharing mainly strings of text. We originally this designed where a new connection assigned an ID by the server. This would have used the random integer library in python to generate a 6-digit number which would then be assigned to the user. This would then be checked against an array (or a list in python) to see if some had been assigned the same ID. This was scrapped in favour of allowing the user to choose an ID, which is what is currently implemented. This is done by having the user choosing an ID when they connect to a room, a feature we will discuss later, and this is checked to make sure every ID is unique. This is better as it allows for users to choose their name so they can be more identifiable to the other users. We also managed to successfully implement the coordinator, when a user connects, it checks to see if they are the first user, and if they are, they are given the coordinator role. This can change if the coordinator leaves, the next person is selected to be new coordinator. As specified in the specification, the coordinator can see the IDs, IPs, and ports of the clients that are connected. Other than the implementation, in which we have used lists and nested if statements when the conditions for assigning a coordinator are right (first person into a room or the current coordinator has left), there isn't really anything to justify as it was a required function of the project. This can also be said for the sharing of IPs, IDs, and ports from both the server and client as it was a requirement and was implemented by using the sockets and client handler to show this information. As mentioned before, we have implemented rooms into the program. This feature allows for clients to choose a room and connect to it and these rooms will be separate from each other and each have their coordinator. Look at figure 1 below, it is a flow chart we produced around the time we started implementing the rooms feature. Figure 1 is a visual description of what the client does. When compared to figure 2 (also below, which is a flow chart describing the server file, it is less complicated, but you can see the function of the rooms and how it has been implemented to the process. This once again was implemented by using lists with the same code that assigned the coordinators does it but takes the relevant group into the assignment. When coming up with a solution We examined multiple different ways to approach the problem, one example was to assign an IP address to each group, such as that if a user wanted to create another group, this would mean assigning a new IP address to that group and starting another private server linked to that IP,

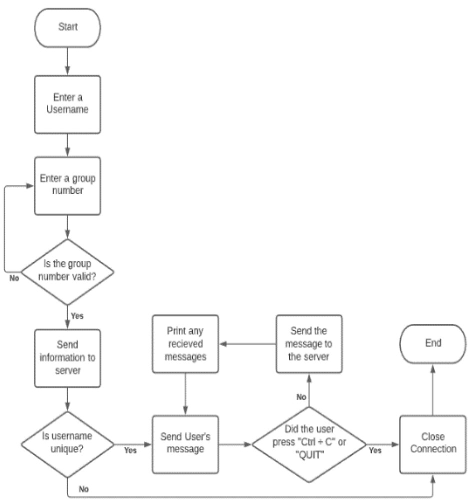
Start
Enter a Username
Enter a group number
Is the group number valid?
No
Yes
Send information to server
Is username unique?
Yes
No
Send User's message
Print any recieved messages
Send the message to the server
Did the user press "Ctrl + C" or "QUIT"
No
Yes
Close Connection
End

Figure 1: Flowchart describing the client code.

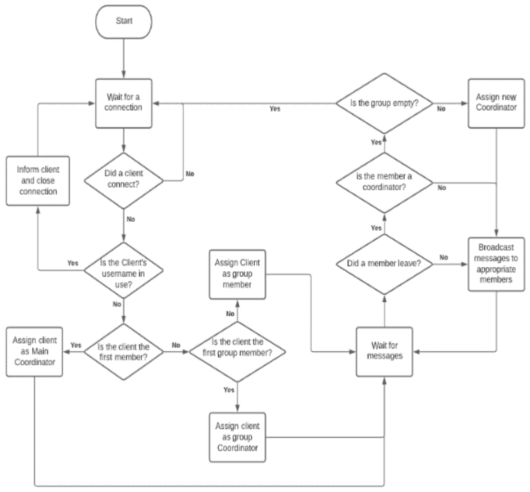
Start
Wait for a connection
Did a client connect?
No
No
Inform client and close connection
Is the Client's username in use?
Yes
No
Is the client the first member?
Yes
Assign client as Main Coordinator
No
Is the client the first group member?
No
Assign Client as group member
Yes
Assign client as group Coordinator
Wait for messages
Did a member leave?
No
Broadcast messages to appropriate members
Yes
is the member a coordinator?
No
Yes
Is the group empty?
Yes
No
Assign new Coordinator

Figure 2: Flowchart describing the server code.

we then realized this would not be ideal as it would increase the amount of network traffic and increase system space and processor usage. This is because it meant having to open another server each time a user wanted to create a new group, meaning less power was focused on the main server. This also meant that there was a limit of groups we could maintain, as it was crippled by the number of IP addresses we could open. So, this solution was clearly superior. This expands the program from the server being an individual room, to the host up to 254 rooms and is overall a welcome feature as it allows for multiple cohorts of people to use it under the same server. A feature that could be added down the line to improve this would be to people able to add a password when the room is first created, the co-ordinator would have access to this password and able to change it. This would allow access to a room to be restricted allowing privacy. After all the users leave, the room resets and the password is removed. This is food for thought and a possible feature to add in later iterations. A big portion of implementing these features was using a UML diagram that we had produced. Figure 3 shows how we implemented the connection by having one main class of client that sent over the important information, the ID, IP and the port.

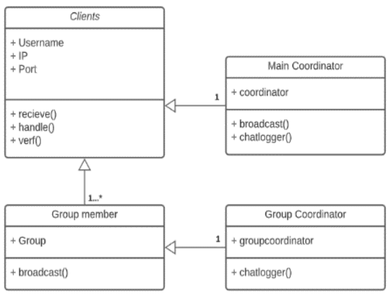

Figure 3: UML describing the clients.

The information is then enhanced further by grouping members into a group attribute and the coordinator which applies the coordinator attribute to the relevant client.

Another small feature we added is the ability to send the message ‘QUIT’ to leave the room. This was implemented by seeing if the message was the same as a phrase using ifelse statement. The allows people to quit with text but you can imagine people might leave when they type quit, so it doesn’t account for the context of the conversation. Pressing ctrl + C also lets the person leave which was a requirement and when a person has left, we have implemented the feature in which others are notified by the departure of one of their group mates.

Analysis and Critical Discussion

In this section we are going to thoroughly examine the code after it has been executed and make analysis based on all implemented technical tasks. In the second part of this section, we made a critical discussion for each part of the code and its implementation. The topics discussed will include the modularity using design patterns, fault tolerance and JUnit testing of the program in a component-based development. Modularity is a concept of re-usability, flexibility and minimization of duplication, it represents the process of creating multiple modules which can be combined and linked into a complete system. The program contains many modules to help do this. In the "server" code the first module is defined with the name starter, it allows for two inputs - the host IP and the port, and it also defines and sets the values for the global variables that will be used throughout the program: hosted, poster and server. The second function defined is called gettime this is a simple function used to get the current server's time to help log and time stamp messages sent and stored on the system. Then a third function is defined called chatlogger, this is a simple function which takes a message as an argument and writes it to a new text file, this is to allow the server to store any messages sent by the clients on the system. The next function is broadcast, it takes in the user's message as an argument, processes and decodes it using ascii, then it sends it to all the appropriate members, based on the group number linked to the message. The function splits and then stores it as a local variable to help connect the broadcast to the appropriate group. To increase fault tolerance, we included a try method to store the message in case it had the group identifier. For every client in the client list, it irritates and send the messages one by one, then it compares the group number in the broadcasted message with the groupnick list - if they are equal that means the message is sent to that user. We use chatlogger to log the messages sent from the server by calling it with the message sent as an argument. The next function called handle is included in the code helps to handle all the messages sent to the server from each client and if the client abruptly disconnects from the server instead of sending a message, the client will be removed from every list. Another part of that function is that it removes and replaces the group coordinator automatically with the next user who connected after the current one, in case the user is the group coordinator for the specific group. The next function we created is called receive, This function allows the server to receive and handle multiple connections whilst sorting and storing all the relevant information about the connections into the appropriate groups whilst checking if the username chosen is already in use, it then sends the user the currently connected clients, their groups, Ips and ports as well as the main coordinator in the server, it then initiates threading with the handle argument to allow the server to handle multiple messages. In the second source code called "client" There are two main functions, the first being receive which allows the client to deal with the messages sent by the server, as well as handling any errors by printing out a more user-friendly error before closing the connection. The second function is called write which allows the user to input a message to send, the function then converts this message into the correct format so that the server can digest it more easily.

For this module because we are using a client and server to communicate, the fault tolerance is based on the software and not hardware. Fault tolerance is about how a system or software can detect errors and continue working despite these issues to a serviceable standard. This means that instead of being fully dysfunctional it will alternatively continue working but not as efficiently as it would if there were no errors. Ultimately the purpose of a fault tolerance is to “rather than a complete system failure, a failsafe design allows for one or more segments of the system to operate as the replacement of the failed part is sought.” (importance implementing fault tolerance system, n.d.)

When it came to the testing phase, we were able to correctly create a messenger application. Users could also join groups as can be seen in the screenshots. After a user input their nickname, they are prompted to pick a group number between 1-254, before connecting to the server. By default, each group has a coordinator (has the role of being an admin). Another requirment which can be seen is, whenever a new member joins or leaves a group, in the chatbox the other users are notified of this action by a broadcast sent by the server. Additionally, the chat only allows for unique usernames, this is to make it easier to distinguish them in the logs. We faced a problem regarding the requirment of handling user’s who are coordinators. Proper disconnections: through the testing, to make sure the that the appropriate action happens when the group coordinator leaves, we came across an error where every other member becomes the coordinator. We solved this by breaking the if statement on line 106 after one iteration to allow for only one coordinator in the group. We believe the program should run sufficiently, due to our testing where we made sure that if a user were to disconnect the fault tolerance would still be high and the program would still function properly with no errors.

Conclusion

To conclude this project, we have learnt a lot from advanced programming. As mentioned in the introduction, we had already had experience with python, but advanced programming has taught us some new concepts and ideas to when it comes to critical think and problem solving, two things that are imperative. The code that we have written is, as far as the specification goes, it’s pretty much feature complete and these features have been implemented with ease. We tried to have our code as compartmental and modular as possible but using functions. We have also commented our code as to help people looking at it understand better as well as to show that we have an understanding of what we have written. Some features we could have added were rooms that had greater security with the passwords, and a clear but intuitive GUI, however the command line works fine. We tried using Microsoft teams and discord text channels to communicate, and while it was a requirement, we couldn’t commit to it and ended up just using discord voice calls to talk to each other as well as share our screens to show the code we had written and to get help, we didn’t record these calls, but we found doing it all by text was far too inconvenient. To improve we should have recorded our communications maybe as audio files. We also had inconsistent pacing, we start early but other modules and commitments made it to where we would take week breaks and it would break up the workflow so to improve, working on time management would be a good idea. However, overall, we should be proud of ourselves as we worked together as a team to make a fairly feature complete product.

Bibliography

importance implementing fault tolerance system. (n.d.). Retrieved from safebytes: https://safebytes.com/importance-implementing-fault-tolerance-system/