Mini Project 1 – Simple File Sharing

Overview

The object of this practical was to build a simple file sharing application (SFS app) which would use TCP protocol to send files across a network to other instances of itself. The practical was split into 2 different parts, a Client and a Server part. The Server was set up to listen on a port and, when a client connected to it, send files over the network to it from a pre-entered directory the user entered. The Client entered an IP address to connect to the server, and then receives a list of files from the server, and enters a file which they wished to access and then they downloaded the file to the pre-entered directory. The initial specification was to only transfer simple text files. The program also used basic error handling in the event of either a client or a server going down.

Certain extensions were added as follows to the basic specification of the code.

These included adding the ability to transfer all file types, the ones tested included the basic text files, and JPG files, PNG files, XML files and .java files, from the server to the client, such that any file which was present in the allocated directory could be transferred to the client. One more extension which was added was that the user was able to find all other copies of this program running, and inform the user that they are ready to share file. Another extension which was added was the ability for the client to request another files from the same server after it had received the first one. Furthermore, there was also the ability for the user to be able to connect to another server after they had finished getting files from the first server. An exit string was also added to the program, which allowed the user to be able to close the program at any point while properly closing all of the connections. Input validation was also added to ensure that the user could never send nonsensical data to the user, other than when they were entering the file name.

Design

The design of the program was in 3 parts; the first part was the starter program which started the Server thread, and also started the Client Program. This took in 2 arguments, specifying the upload directory for the server and a download directory for the Client part of the program. If no arguments were entered, a decision was made for both the client and server to operate out of the directories they were started out of. If incorrect arguments were entered, e.g. if the user entered in Hello world as a directory, then the program rejected the arguments and returned an error.

The second part of the practical, was the Server side. The server was set up in a separate thread, to allow it to run freely without interference from any other part of the program. The server then listened on a port for any clients connecting to it, and when a client connected to it, the server then started a separate thread to handle the connection for that client, to allow the server to accept multiple Client connections, and then the server went back to listening on the port for other connections. When the Server was started, a separate thread was started to send the IP address of the server to the MultiCast group.

The connection handler used a basic protocol to transfer files across to the user, the protocol was as follows:

Server: SENDS “Here is a list of files”

SENDS List of Files in the directory specified

SENDS “Please enter the file name you wish to access”

GETS the name of the file the Client wishes to access

The server checks for the file in the specified directory, if it finds it, it sends the file size and then the file across the network, otherwise it sends a suitable error message to the user.

SENDS “Do you want to access more files Y or N?”

GETS the answer from the client, if the answer is Y, the handler does the loop again, otherwise it closes the connection.

CLIENT

GETS The list of files in the directory

SENDS the file name to the server

GETS either an error message from the server: “Files not Found” or gets the length of the byte array which contains the file, if the file length, then reads as many bytes as there are given in the length of the byte array, and writes it to file.

SENDS Y or N as to whether they want to access more files on that server.

Repeats the process to allow the Client to download more files from the same directory in the network.

The Client is the third part of the program, it had two separate parts, the first part was the thread spun off when the client loaded, which checked for DataGram packets from the MultiCast group, and added the IP addresses contained in them to a list of known datagram packets, while filtering out the Datagram packet sent by the Server part of the Program.

The main Client asked the User which IP address they would like to connect to, having been given the list of IP addresses which are sharing files on the network, and after that, the program tried to connect to that Client, if unsuccessful, there were given a suitable error message and then asked if they would like to connect to another IP address and asked to input it. If the connection was successful, the file sharing protocol as described above was called to facilitate the transfer of files from the server to the Client.

Implementation

The implementation was split up into 6 classes as follows. The port the Client and the Servr were working on was port 12345.

The sfs class was a main class which started the Client and the Server parts of the program.

The Server class was the class which set up the server on a ServerSocket, to allow TCP transfer of data to allow reliable delivery, and listened on the port for connections. This also started the threads for the connection handlers.

The ServerMultiCast class was responsible for sending the DataGram packet containing the IP address of the server to the chosen IP address and port in the MultiCast group (228.5.6.7 on port 2345). A packet was sent every 5 seconds, a time interval which was chosen as it was far enough apart to reduce the load on the network and close enough to ensure that when the IP is read by the client, the server is still active.

The transferFiles class was the class responsible for the handling the connection for a client who connects to the server. This was run in a separate thread every time a client connection was made to

the server, to allow the server to process multiple requests. An algorithm was used to find the directory specified and for each file found in the directory, the class sent the name to the Client. The main algorithm for transferring files worked as follows: an attempt was made to find the file in the right directory. If this was successful, then a byte array was created with the same length as the length of the file, and then using a buffered input stream, the file was read byte by byte from 0 bytes to the byte length of the file into the input stream. After that the array size was sent to the client as that would tell the client how many bytes to read from the when they were reading the file. The file was then written to the output stream, again byte by byte from 0 to the size of the file. The program then read the user input as to whether they wished to access more files and repeated the loop. If at any point the exit string "exit" was entered, the handler exited the loop and then called the clean-up method which securely closed all of the socket inputs and outputs and then closed the socket again. If errors occurred at any point throughout, the clean-up method was called.

On the Client side of the program there were 2 classes, the Client MultiCast class and the main Client class.

The Client MultiCast class, ran in a separate thread, where it joined the MultiCast group defined in the ServerMultiCast, and received packets every second, stripped out the empty spaces in the string and added the IP address it received in the Hash map as the key, along with the date that the packet was received, it did not do this for the IP address of the other part of the program which was running the server side. Every 5 seconds, the program iterated through the Hash Map, and got the current time, and compared it to the time that was stored in the Hash map for each IP, if the difference was more than approximately 20 seconds, the program removed it from the Hash Map. The reason it did this was that every time the program received a UDP packet from an IP it already had a packet from, it overwrote that entry in the Hash Map with the current date, meaning that the date the IP was received was being updated every time a packet was received from that IP, and since a UDP packet was sent from the Server every 5 seconds, the program would only remove an IP from the list of IPs active on the network if it had not received a packet from that network in more than 20 seconds, or at least 4 cycles, which worked well as a buffer to allow for dropped packets, or packets being lost, but still gave a reasonable time for inactive servers being removed from the list.

The Client class was the user side of the program. It allowed the user to be able to connect to whichever IP address they wanted to, and if it was not possible on the IP provided on the default port, a suitable error message was thrown, and they were then given the option to reconnect to another IP address. The Client also gave out a list of active users on the network by iterating over the HashMap and then printing out the user's IP addresses, or a suitable message if there were no active users. If the user successfully entered in the IP and connected, they were taken through the protocol described above to read files from the server. If at any point, errors occurred while connecting, or reading from the server, a clean-up method was called which closed all of the input and output streams, and then closing the socket connectivity, and then going back and allowing the user to connect to another IP address.

Adapted Code Fragments

Basic code was adapted and used from various places, the Client Server Class structure, having different files for Client, Server and the connection Handler was adapted from the Client Server example in the Examples folder in the CS2101 folder. The algorithm for transferring files was also adapted from different file transfer algorithms found online in Stack Overflow, and a website name as follows:

www.rgagnon.com/javadetails/java-0542.html.

The code for sending the file was very similar to the above link, however reading the file was adapted to allow the user to continue to be able to interact with the server.

Furthermore, the algorithm for finding all of the files in a directory was also adapted from a similar one which was found on StackOverflow on the link as follows:

http://stackoverflow.com/questions/5694385/getting-the-filenames-of-all-files-in-a-folder

The file protocol was written by myself.

Class Diagram

The Class diagram for this program is located in a separate file.

Problems Experienced

Most of the problems experienced in completing this practical were due to small errors, e.g. reading on the wrong port or not trimming the hidden spaces from the string coming in from the DataGram packet, however some of the major problems were mainly around the file transfer, as the file reader would not move on, as initially it was waiting for the server socket to close before moving on, and after that the errors which occurred were due to it not reading the correct amount of bytes.

Testing

Testing screenshots are present in a separate document.

Testing was done in 2 ways, the first testing was done running 2 instances of the program on the same machine, with one running as the server, and one running as the client, and seeing if file transfer took place between the two different directories. The second type of testing which occurred was running the same program initially without multicasting and seeing if the files could be transferred across the local lab network, this was followed by then introducing multicasting and seeing if the MultiCasting worked as expected, with the program not initially finding any computer on the network, but slowly finding them. Also it was tested that if a server was stopped, the other server would no longer show it in the list of users after a period of time. It was also tested if the server was closed on the primary machine/directory, the secondary machine/directory client also closed the connection to it. The exit string was also tested to ensure that it allowed the program to exit cleanly. All of the above testing was run with different types of files including JPG's and text files to ensure that it still worked as expected. Furthermore, the input validation was also checked to ensure that it worked as expected, not connecting anywhere where the connection was not established, and rejecting invalid strings.

Improvements

In a real world scenario, the program could be improved in a number of ways, there could have been a GUI which would have made the program more user friendly. Furthermore, from a more practical point of view, the program just creates new threads each time a client connects, and it would be possible if a large number of users connected, that the program could crash due to the large number of threads active, and a better approach would be using a thread pool which limited the system usage. Also blacklisting could also be done for certain files, as in this scenario, if the user didn't enter any arguments, the directory was set to be the same directory as the client and the server, which would allow the user to be able to get hold of the Class and Java files, which ideally should not happen, thus a blacklist would be much better to stop the client being able to receive the running files of the server.

Conclusion

The Program designed completed all of the basic requirements, and went beyond the basics by allowing the user to be able to share any type of file, and finding all of the running copies of the program in the local network. Functionality was also added to allow the user to be able to request multiple files, one after another. Furthermore, after the user was done requesting files from the Server, they could connect to another server. A very simple protocol was used to transfer the files from the Server to the Client. The program successfully was able to transfer files between 2 versions of itself between 2 previously specified directories, the program was also then able to find other instances of the program and communicate this to the users in the form of IP addresses.