Systems Software Coursework

Group - B - CSM - Team 3
Alfie Dunstan(N0865426)
Mikolaj Bukowski(N0839914)
George Nechifor(N0898052)
Muazzam Chaudhary(N0854544)
Rajen Patel(N0874802)

Introduction

As a team we have been tasked with designing to develop a java-based concurrent client server system for a farmer in order to help him improve the management of his business through the use of technology. As a team we aim to develop the following components for the farmer: digital weather stations controlled by microchips, a client server to be manned by the farmers workers to provide information about specific data connected to the connected weather stations and a central server machine which provides services for both of the stations i mentioned previously.This report aims to provide reasons for any design choices that we made with how they were implemented into the program and explaining the features that go with them.

Features, Design and Implementation

Implemented Features

Here is a list of all the implemented features within the program and a short description on what they do.

Server:

- Distinguishing between both incoming weather and user clients so that the server is able to provide the correct services depending on whether or not a certain client was connected.

Weather station clients:

- Clients that log in are automatically connected to the server and are assigned a specific client ID. When a client disconnects from the server, the ID assigned to them is freed up.
- Updated information gathered by the weather station using readings from its sensors is continuously sent to the server

User client:

- Users gain access to the server via a login system which is password protected to ensure that only authorized users are accessing the server.
- Clients can obtain up to date information about the field
- Clients are able to see the the currently connected weather stations and their respective ID’s

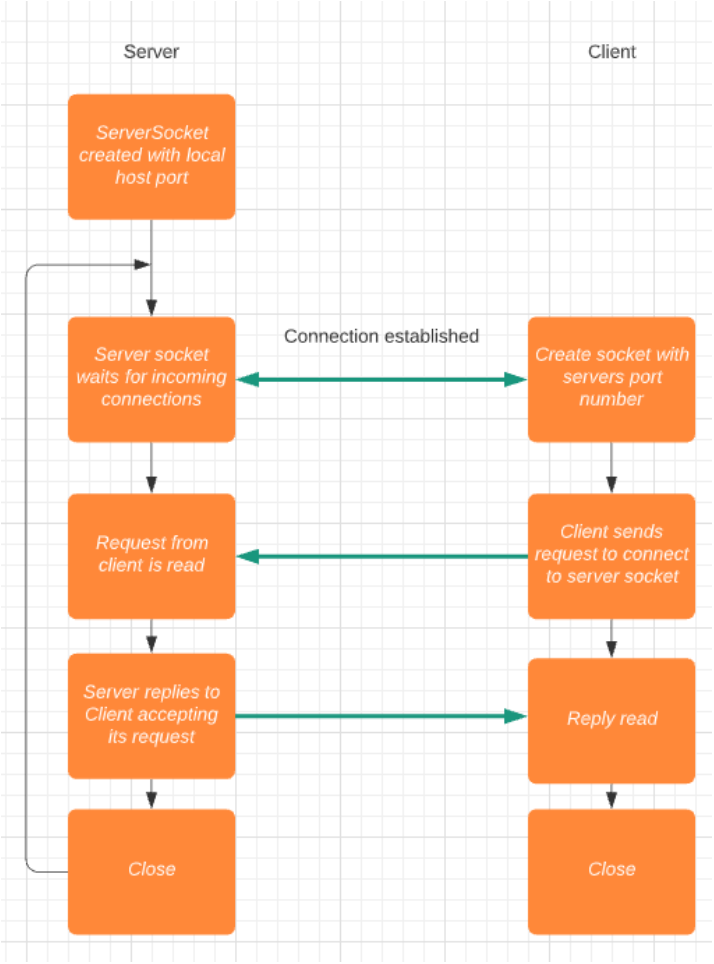
Protocol Choices and How Sockets Would Be Used

One of the first decisions that we had to make as a group was to either use the connection oriented protocol(TCP) or the connectionless protocol(UDP) or more specifically, the decision to make use of stream sockets or datagram sockets.

The first aspect that we decided to look at was the speed of both protocols. Out of the two, UDP is the fastest at sending data as it does not have any form error correction meaning that there is no automatic confirmation to the originator that the packet has arrived at its destination or if the packet had been corrupted or interfered with during its transmission process. Another reason as to why UDP is faster than TCP is also due to the header size of TCP being larger as it has more details such as the sequence number,acknowledgement number and error detection.As a group, we decided that the speed gained by using UDP is simply too negligible for it to be much of an advantage over TCP. If we were dealing with things such as video or audio being played then UDP's speed might have been preferred however in this instance we are only sending over characters and integers so the speed of TCP is more than enough.

The next aspect that we would base our choices on was the reliability of both protocols. Due to the fact that the main functionality of the user client is to obtain the information gathered by the weather station clients, it is hugely important that the data actually gets to the clients and is not corrupted or lost along the way.Because of this, TCP would be massively favoured as unlike UDP it has automatic error correction. If a packet that has been sent is found to be corrupted or just simply has not reached its final destination then TCP does not send an acknowledgement for that packet and as a result, the sender is prompted to resend it and by using this the message is sent free of errors. While UDP does have an error checking mechanism in the form of checksums, it does not attempt to resend it and will consequently end up discarding the packet. Another advantage that TCP has over UDP is its ability to ensure that data is received in the same order that it was sent. It does this by adding a sequence number within its header which ensures the recipient is able to arrange the original sequence back.

With all of this considered, our chosen socket to use was the TCP stream sockets as their seemingly biggest disadvantage of speed compared to UDP is not an issue for the distributed system we are making.With the decision out of the way we were now ready to actually implement the stream sockets for our server and clients and think about how they would be used.



The diagram above shows the basic setup of designing a client and server that communicate with each other via TCP sockets. To begin with, a server socket is set up and would usually wait for incoming connections indefinitely. In our case , we have specifically set it up so that it only accepts connections for a maximum of 5 different clients. Happening at the same time on the client side of things, another socket is set up with the hostname and servers address which in our case is ” localhost” with port 9090 and is then accepted by the server thus completing the three way handshake. From here both the client and server are able to send messages to each other by making use of input and output streams using classes from the java.io package. With this in place, we now have a basic system of client - server communication and can start focusing on implementing multi- threading and have multiple clients to a server.

How Multithreaded Client Handling Was Implemented

The next step in making our concurrent server client system was to handle multiple clients at once through the use of threading. Within our code we included a client counter which is located within the start() method which has the role of keeping track of how many clients are currently connected. This means adding to the counter if a new client of any type is added and decrementing the value when one is removed. The start() method works by always trying to accept any incoming client requests and would usually be waiting for them indefinitely. However, we have made it so it only waits for requests so long as the number of clients currently connected is less than 5. Within the client counter we have included a method called addClient() which receives any accepted connections and then creates a new thread by using the client handler method and then stores these connections and threads within a vector in order to keep track of them. We decided to store things using vectors as we were the most familiar with them over alternatives such as arraylists. Next , the clientHandler class shows exactly what the threads will be doing. In our case, the threads will be differentiating the different clients and then will perform the correction action based on whether or not a certain client was connected.When the connection is dropped , a message will be printed saying that this has happened and then the thread will stop upon reaching the end of run() and the number of clients is decremented and that client ID is freed up for a new potential client to be ran.

Additional Features

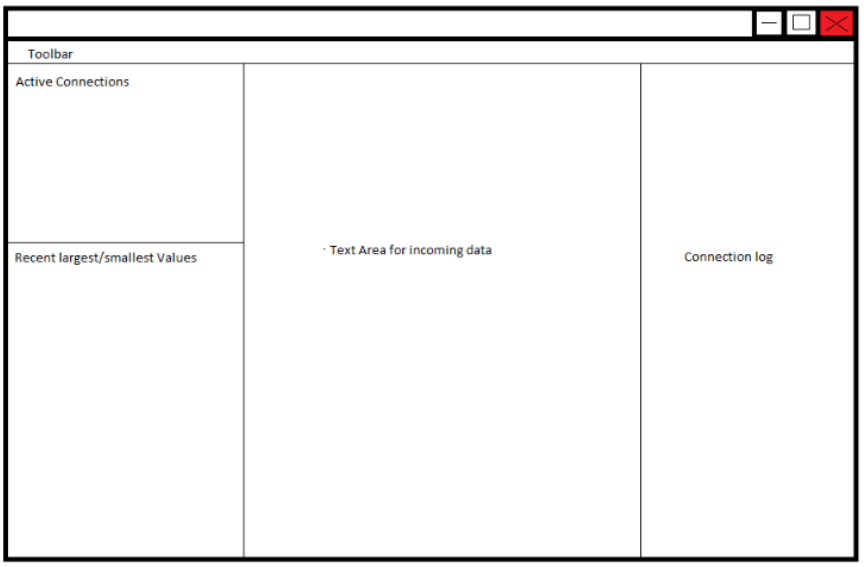
The server includes a data handler class to manage any weather data coming in and going out. It stores the latest weather data inputs in a queue format such that whenever the user client is offline, it prepares the latest data that it missed on. When the user client does eventually log in, then all the recently stored data will be fetched from this queue and sent to the user client as a part of history / missed out data.

Conclusion & Potential Future Work

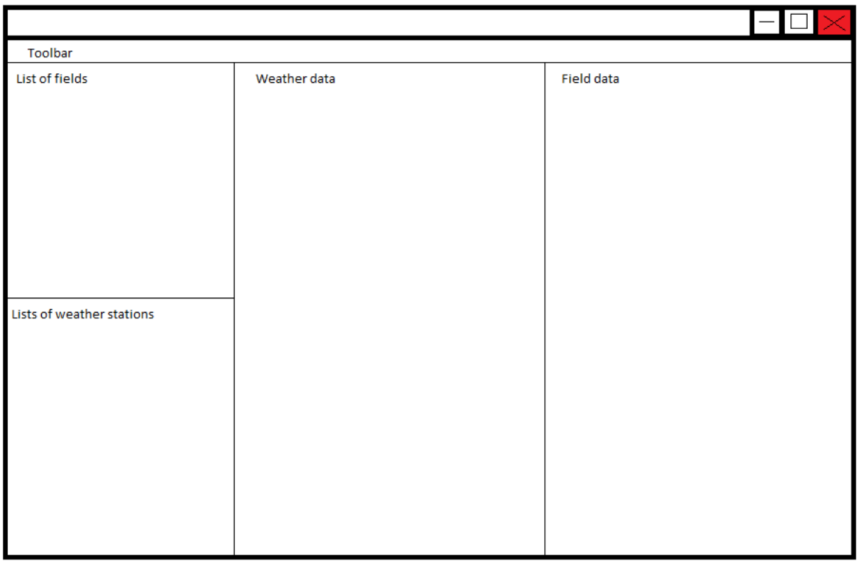
By the end of the deadline , we unfortunately did not manage to achieve everything that we would have wanted to include. While we did manage to implement the main and fundamental features required for a concurrent server client system we did fail to fully incorporate a completely functional GUI and there were some more advanced features regarding the user client that we would have liked to add given the opportunity of more time.

GUI - when it came down to the GUI there were display issues with the frames that were being produced alongside the server and clients that were running. These display issues would be something that we would have liked to resolve if given more time, but also being able to add more to the GUI so that it wasn't too simple where it looked as though it was lacking much user interaction. With these additive features of the GUI in mind what i would specifically have done, is to add some colour to the frames that were created to it wasn't to blank and boring, I also would have added some images or graphs/diagrams to help give a more visually pleasing aspects to the GUI for the user to look at. One other thing I would have done is to create a GUI for the Weather station as well to help present the data about the fields and weather that are being transmitted to other user clients.

Server GUI design:



Weather Client GUI design:



One of the features that we would have implemented into the main program given more time is the ability to store the weather data in files and then allowing the user client to request a download of these files, potentially including extra and more detailed data. This way the user client will have an option to receive a larger set of data, allowing for a more in-depth analysis of field data. This could potentially be achieved by storing all of the data gathered by the weather station client within a new text file and time stamping when that data was gathered. For the user client, we could have a button for each weather station client that when clicked will read from the corresponding text file which can then be presented / downloaded to review and look at .