All To Largest resource Allocator For Distributed Systems

Andrew Campbell - 45638292

Jack Mahon - 45291896

Jake May – 45436126

Introduction:

The aim of this project is to devise a design based around an existing simulator (ds-sim simulation protocol) with the goal of implementing a script capable of dispatching jobs from the client-side of the architecture. The ds-sim simulation protocol describes a protocol that inter-connects the client with the server-side simulator, as well as receiving and scheduling jobs.

Stage 1 of this project is focused on applying the client-side simulator to utilise a job dispatcher whose job it is to allocate resources by way of selecting the largest server that is the first of its type. This is achieved by selecting the lowest possible ID of the server type that is available.

In this instance, the largest server type is determined by the number of CPU cores available, the higher the number of CPU cores, the larger the server.

System Overview:

The client-side simulator is tasked with obtaining jobs which are read by the server from an XML document and scheduling the tasks based on prior-made requirements. In Stage 1, the client-side simulator is created to read jobs from the server and schedule them to the largest server. The largest server in the case of Stage 1, is the server that has the most CPU cores.

The interaction between the server and the client begins with the client connecting to an open port in the server (port 50000). The client then sends “HELO” to the server which then responds with “OK”. The client then responds with an authentication message with the name of a user (the current username for the account being used on the computer running the client) “AUTH name”. The server then responds with “OK” to acknowledge the authentication.

The client then tells the server that it is ready to take a job by sending “REDY”. The server then sends a job to the client and the client asks to see all the servers that can be scheduled to. The client then looks at each server’s CPU cores and chooses the one with the most cores (the largest server) and schedules the job to it. This process repeats until “NONE” is sent instead of a job by the server, at which point the client sends “QUIT” and quits the program and then the server sends “QUIT” and also quits the program.

Design:

Design Philosophy

The overall design philosophy consisted of using functions to make the code cleaner and more modular. Extensive comments have been added for better developer readability and documentation.

Considerations/Constraints

During the design of the simulator, it was imperative to understand where the data was going to be coming from, as well as the programming language it was in, in order to access and utilise it properly. In the case of this project, data was only required to be read from the server in Java. It was required that the “-n” command was used to read the end of each line in each text file on the server.

No other commands were required to be read from the server apart from “JOBN” commands as per the project specifications.

Component Functionality

The functionality of several different components is apparent in this simulator. Separate functions inside a while loop have been utilised. Port 50000 has been chosen as the port for the client and server to communicate over.

There are two separate functions defined for the sending and receiving functionality of messages for the client:

*readMsg*

Stores the incoming data that is received via a data input stream into a byte array that is then converted to a string and printed to the console.

*sendMsg*

Sends a message that is first converted to a byte array and sent to the server via a data output stream.

Implementation:

Three external libraries are used in our implementation of the java client. These libraries have the following addresses:

* Java.io.\*
* Java.net.\*
* Java.nio.charsets.StandardCharsets

Each of these libraries is implemented to facilitate some aspect of communication between the client and the server, the specificities of which are discussed below.

The generic java input output library (java.io. \*) facilitates input and output capabilities in a java program. In the client, the library is used to implement the DataInputStream and DataOutputStream objects, as well as the corresponding exception handler (IOException). The DataInputStream object is used to receive a byte array sent from the server to the client whiled the DataOutputStream object is used for the inverse; to send a byte array back to the server. All implementations of either of these objects are encapsulated in a try/catch block, making use of the library's exception handler.

The generic java network library (java.net. \*) facilitates network capabilities in a java project. In the client, the library is used to implement the Socket object. The Socket object allows network connectivity to specific machine over a specific port, in this case the client connects to the local machine over port 50000.

The java charset library (java.nio.charset.StandardCharsets) facilitates encoding and decoding of byte arrays into different formats. In the client, the library is used to encode and decode between byte arrays and Unicode Transformation Format 8-bit encoding (UTF-8). This is used to make the byte arrays human readable.

The ds-sim control protocol is broken down into 14 steps, in our implementation of the client these 14 steps were broken up and distributed over 3 primary functions and 2 helper functions, listed respectively:

* public static void main (String [] args)
* Public static void handshake (Socket s)
* Public String [] findBiggestServer(String currentMsg)
* public static synchronized String readMsg(socket s)
* public static synchronized void sendMsg(Socket s, String currentMsg)

The helper function readMsg is used to accept the byte array from the server, converts it to a String and returns the String. The function accepts a Socket as an argument and uses the DataInputStream object to receive the byte array sent from the server. Once this byte array is received it is converted to UTF-8 encoding and returned. If no byte array was received or an IOException occurred the String “FAIL” is returned to indicate failure. The sendMsg function is used to take a specific String and send it to the server. The function accepts a Socket and a String as arguments. The Socket defines the port and machine the client is sending the message to. The String is converted into a byte array and sent to the server. Both the sendMsg and readMsg functions are synchronized to avoid any race conditions in reading the input and writing the output over the same port.

As is standard, the main function is the first function implemented in the client. After initialising and instantiating the necessary data structures the first function called is the handshake function which handles the first 4 steps of the ds-sim protocol. Firstly, the client sends “HELO” using the sendMsg function. Then, the client receives the reply from the server using the readMsg function. Upon receiving “OK” from the server the client replies with the String “AUTH xxx” where a username supplements the “xxx” portion of the String. Finally, the client receives another “OK” from the client and the handshake is complete.

The main function continues the protocol upon completion of the handshake. The client then sends the String “REDY” to the server and iterates over a while loop until receiving a response. If the received message contains the String “JOBN” it is split at every space and each segment of the String is stored on a String array. These segments after “JOBN” represent the incoming job’s: submitTime, jobID, estRuntime, core, memory, and disk. The client then sends the string “GETS Avail” followed by the previously identified core, memory, and disk attributes from the job specification, each separated by a space. The server responds with the available server's data and the client responds with the customary “OK”. This data is then passed the findBiggestServer function.

The findBiggestServer function accepts a String as a parameter and returns a String array. The String passed to the function is separated by line breaks and each segment is stored in a String array. Two for loops are used to identify the most cores in any server and then the server with that number of cores, subsequently identifying the server with the most available cores for the current job. The identified server’s details are returned in the form of a String array.

The main function then resumes once the server’s details are received from the findBiggestServer function. Upon receiving the message “.” from the server, indicating there is no more information to send, the client sends the String “SCHD ” followed by the jobs jobID and the biggest server’s type and ID, indicating which job to schedule on which server. If the initial message received after the handshake is “DATA” instead of containing “JOBN”, the client simply responds with “OK”. After the protocol is complete the client sends the String “QUIT” and receives the same String from the server. The client then exits, and the operation is complete.

The design and implementation of all the code can be attributed primarily to Andrew Campbell who achieved a successful implementation of the java client. Any contributions made form the other team members were minimal and they were more involved in the documentation portion of the assignment.

References

Public Assignment Repository: https://github.com/Andy1184/COMP3100-Assignment-1