Project title

**Basic scheduler with inbuilt job dispatcher within a client side simulator**

Group members

Jonathan Cawood (45887454)

Carlos Larino (45942854)

Introduction (1/2 page)

The aim of this project is to develop a job scheduler for a distrusted system .With specific relation to Stage 1 requiring the design and implementation of a “vanilla’ version of a client-side simulator that includes a basic scheduling function with a simple job dispatcher. The simple job dispatcher upon successful implementation will send all jobs to the first one of largest server type. The largest server type will need to be determined in order for this process to work. The role of a job dispatcher within a distributed system is key for ensuring the efficient use of computer systems including distributed systems which will be highlighted within this project.

System overview (1/2 page)![Diagram

Description automatically generated with medium confidence]()

Design (1 page)

**Design philosophy:**

The design philosophy of the client-side simulator that would act as a simple job dispatcher. This client job dispatcher will connect to a server-side simulator, receive a job and schedule them for the intention of efficient usage of the server. The client-side simulator should have basic scheduling functions and dispatch jobs efficiently. Our group can achieve this by ensuring that the largest server simulator is found between the communication with our client and server, then using that large server to schedule and dispatch jobs into it. This allows jobs to be ran into one large server rather than having jobs be dedicated into smaller server which could be expensive to run. In addition, this could create downtime for other jobs in real world applications.

**Considerations:**

Our group needed to consider that the server will schedule the jobs according to the largest server. However, this can only be done through client-side simulator code with no changes to the server code. To find the largest server within the distributed system, our group needed to consider that there will be multiple servers to choose from. Hence, creating code to find the largest server and inputting all the job requirements into that server is a must. Our client dispatcher should have the intent of efficiency, not only mitigating redundant code but also during communication and job dispatchment between the client and server. The communication between client and server should not have unnecessary messages between them. Furthermore, the client should be able to establish communication by running a handshake between the client and the server. The client should be able to send and receive messages to the server whilst the server should be able to do the same towards the client.

**Constraints:**

Major constraint of our design was that our group lacked the experience to create a client job dispatcher. Members of our group have never created a simple job dispatcher before. Although interesting to learn, hard to implement. Hence, understanding how the ds-sim server protocol received and categorised jobs were difficult to implement. Implementation of various classes, functions are needed to ensure that the server receives the proper transmissions/ Jobs.

**Functionalities of each simulator (Client-side Job dispatcher):**

The client-side job dispatcher will need to establish communication with the server-side simulator or any server in the future by running a handshake between them through a “HELO” message. The server should be able to receive this message and the client should be able to read the message received by the server. The client should also be able to authorise a user to the server and once the server authorises the user, both client and server should be ready (“REDY”) to send and receive jobs. The client will be able to send queries on what the server should do, in the case of the dispatcher, to request all information about all server regardless of their state, ie. Inactive or unavailable (“GETS ALL”). The client should be able to allocate memory to store the data of the server details to be used later. Once it gets all the information about the server, the client-side simulator needs to determine what is the largest server by using a function to determine the “core Count” of the server. Once determined the client should request, he jobs that it wants to be done, then the server should schedule it based on the largest server. The client should be able to receive the data from the schedule reply and then has the ability to terminate the connection between the server and client (“QUIT”).

Implementation (2 pages)

Jonathan Cawood worked on the implementation of initial communication to the server specifically dealing with the implementation of “HELO”, “AUTH”, “REDY”, “GETS All” requests and replies. Whilst also implementing the data handling of the servers from the server side as in the creation and inputting of each server into a created array list. Additionally, the implementation of public functions to reduce redundancy within the code was implemented.

An Array List was used to store all the information of each server using the created Server class. The Server class was created to be able to import server properties into the class and be able to locate and keep track of the imputed servers, such as the ID, State, Start Time, Cores, Memory, Disk. Therefore, enabling the use of a for loop to loop through and determine which server had the highest amounts of Cores with the use of an if statement to compare the core count with the highest core count so far, allowing for the highest to be determined.

The public function Msg sender was constructed to reduce redundancy within the code, as the sending of the message from the client to the server was repeated with only the message to be sent altered. Therefore, an overall function could be constructed that was passed the message as a string to then convert the string to bytes then flush the inputted message to the server. Whilst also letting the client-side operator know that a message has been sent. This feature allows for easier debugging in the future. The addition of this function allowed for the minimisation of code and allowed for the workflow to be easily identifiable.

Likewise, the introduction of the Msg converter and Msg receiver are to reduce the amount of redundant code within the file, with the Msg receiver reading the input stream and writing it to a byte array, to then convert to a string which is then printed on the client side, likewise with the Msg converter reading and storing the input stream, converting the byte array into a string to then return the string to the call.

The three functions were designed and developed to reduce the reptation of code whilst allowing for the ability to read the code sequentially and understand the process undertaken, therefore was designed specifically to streamline code.

References

Cawood. J. Larino. C. Comp3100\_Group26 [online]. GitHub. <https://github.com/jonathancawood/Comp3100_Group26/>

Lee. Y. (2021). ds-sim[online]. GitHub . Available at: [<https://github.com/distsys-MQ/ds-sim/>]