Cloud Job Dispatcher

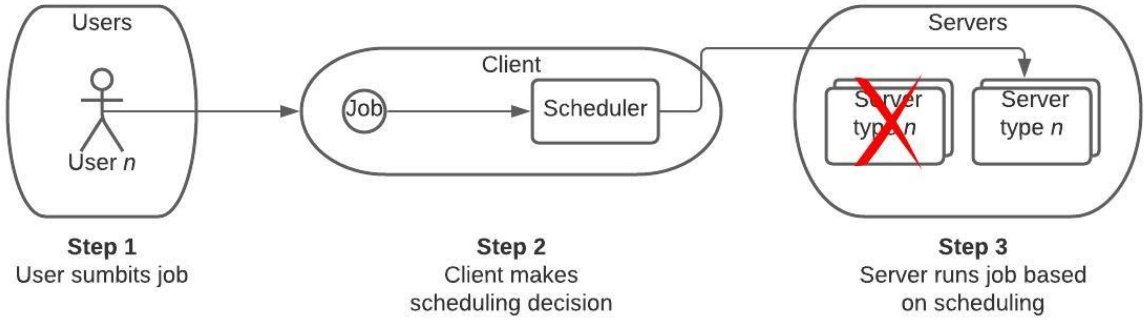
Group Members:
- Jack Davenport (45946396)
- Lucas Turnbull (45947155)
- Elliot Holt (45955964)

Links:
GitHub: https://github.com/mq45946396/comp3100-project

Introduction:
This project focuses on the design and implementation of a simple job dispatcher using a basic client side simulator. This acts as the first stage of a two-stage project that aims to develop a job scheduler for distributed systems. The main task for this stage is to design and implement a 'vanilla' version of a client-side simulator. Particularly this stage aims to implement a predefined ds-sim simulation protocol. The protocol in question connects the server-side simulator, receives jobs and schedules them. This stage also implements a simple job dispatcher that will send all jobs to the first one of the largest server types.

System Overview:
The distributed systems simulator is an open-source, configurable simulation. Its purpose is to act as a plain simulation of job scheduling and execution for distributed systems. The ds-sim uses the client-server model, meaning there is a ds-client and a ds-server component, acting as the client-side and server-side simulators respectively. In this project, the client-side acts as a job scheduler while the server simulates everything else for job submissions and job executions. Since the focus of this project is the ds-client, it is a component that can be implemented with set instructions for job scheduling. As long as the ds-sim simulation protocol is adhered to the ds-server should be able to receive and run what the ds-client sends it.

The simulation follows a simple step-by-step process. The ds-client receives a job generated by the ds-server. The client makes a scheduling decision based on finding the best possible server available and sends the schedule to the server. The ds-server then runs the job applying the scheduling decision.



Above is a figure briefly explaining the steps and workflow that take place within the client-server model for the job dispatcher simulation.

Design:

The design philosophy of the client simulator is to have a modular system for handling each general functionality so that a new replacement script could be created and easily plugged into the client for quick future development and prototyping of alterations to the client. This philosophy was borne of the idea that we should make our code to be as clean and organised as possible so that everyone in the group would be able to quickly figure out what all of the code does without needing to ask the relevant author.

The modules of the client are;
Client.java itself, which is the most vital part and is what every other module plugs into, acting as the center of the entire client simulator, hence it bears the name it has. The main constraint with the client is simply knowing what to pass off to other modules and what to do locally in Client.java so that the client simulator can adequately meet our modular philosophy while also making sure that no module needs to have access to anything that is only used in a separate module's functionality.

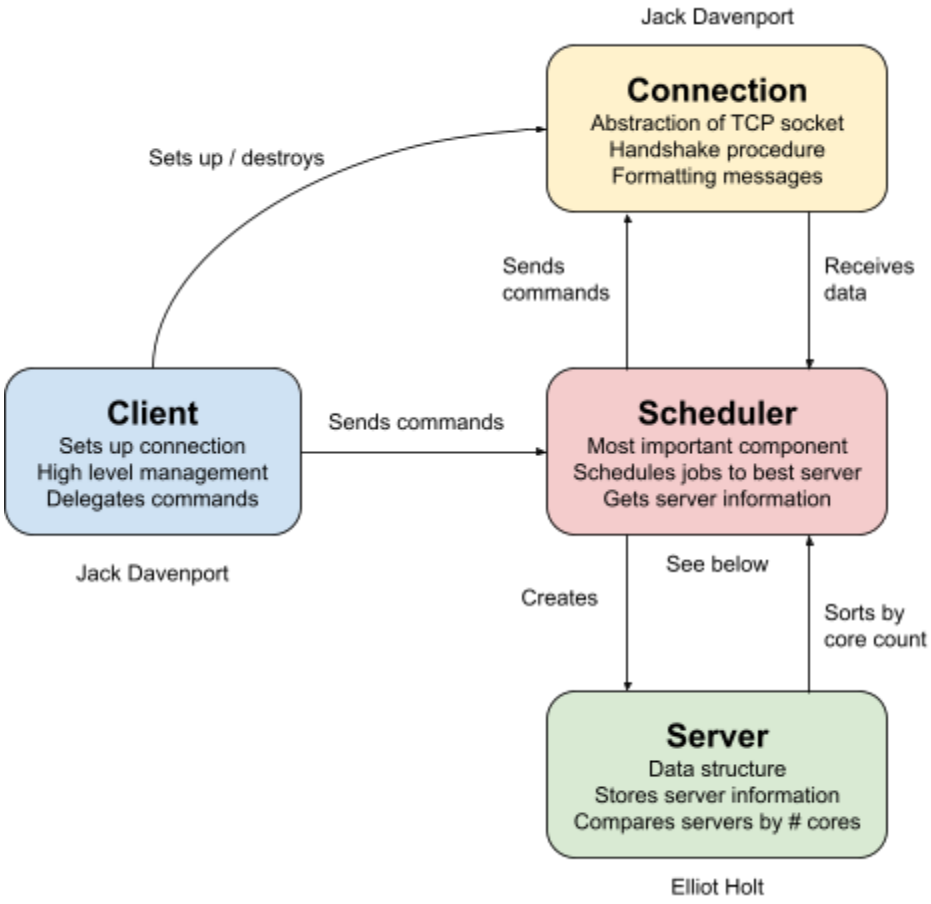
Next is Connection.java which handles the client's connection with the server simulator, going so far as to include functionalities for handling the session with the initial handshake and ending the session but also dealing with communication with the server simulator by sending and receiving messages and commands. The modularity of the client simulator's design was the biggest thing to consider when designing the connection module, since it could have been made to handle everything related to communication with the server simulator and leave the Client class as a simple container simply pointing to Connection.java, but that would conflict with the idea of the the Client class being what connects everything together. This led to the Connection class only being concerned with what goes directly in and out of the client simulator without doing any parsing or formatting beyond turning input and output into a more usable data type.

Finally, Scheduler.java handles the scheduling of jobs to different servers while creating and storing a local version of the list of servers. The modular design philosophy, along with the decision to have the Connection class to not handle the kind of parsing that Scheduler.java requires led to it being tasked with not only the allocation of jobs to servers based on predetermined criteria, but also with the parsing of server data.

There was also a server object created so that Scheduler.java would be able to directly get and set the properties of each server that is stored, enabling faster code execution as well as being a clean way of improving efficiency.

Implementation:

The implementation of the client was a collaborative effort between the three of us. The client is written in Java, and uses TCP to connect to the simulation server. The client is a restructured version of the workshop code, which was rewritten from the ground up to be much cleaner and better organised. This was achieved by splitting up the client into a series of modules, which focus on a specific area of functionality in the program, which helps to decouple the code and make it much simpler and more elegant. Below is a simple flowchart of the modules making up the client, along with who was responsible for each component of the project.



The **Client** is responsible for creating the connection and initiating the handshake procedure. This is also where the hostname, port and username for the server is decided. It then goes into a loop where it continually receives commands from the server. When it does, it will parse the command's arguments, before passing them on to the appropriate module. When the **NONE** command is sent by the server, it will then initiate the closure of the connection.

The **Connection** module is mostly a utility to abstract the socket connection to the server. It allows you to send data to the server, send formatted data to the server (in a similar fashion to **printf()** in C), and read data from the server. It also contains the handshake procedure which is initiated by the Client component.

The **Scheduler** module is the most important to the client's operation. It will receive any **JOBN** command and break it down into parameters, fetch information about each server from the server, then sort them by core count and decide which server to allocate the job to. Currently, it chooses this based on whichever server has the largest core count and meets the minimum requirements of the job. The scheduler is composed of a few sub-functions, which each have certain distinct roles:

Function	Purpose	Author
scheduleJob()	Schedules a job based on the parameters given by the server. It uses getAllServers() and then pickBestServer() to determine the best available server, then sends a scheduling command to the server.	Jack Davenport
getAllServers()	Gets a list of the all servers and their statuses, returning it as an array of Server objects.	Lucas Turnbull
pickBestServer()	Finds the best server in a given array of Server objects. It will also remove any server which doesn't meet the given requirements for core count, memory and disk size.	Lucas Turnbull

The **Server** module is simply a data structure class to clean up the code in the other areas of the Scheduler module. It contains all of the parameters sent by the **GETS All** command and also a function to sort the servers in a list based on the core count, which will result in the server lists being in descending order of core count. Instances of this class are created for every job however, so in future an optimisation could be to keep track of these objects and simply use **GETS All** to update the status of the cached server objects.

Along the way there were several issues we ran into. The first major bug came from the number of available cores on the largest server decreasing as more jobs were added, so eventually a different server had more cores and jobs started going to it, despite the 'allToLargest' behaviour being expected. Therefore the largest server is simply stored at launch and all jobs go to it.

Another was caused because there was not enough bytes allocated to read the server information, so the getAllServers() routine would fail when there were a lot of servers. This was fixed by using the parameters of **DATA** to read more bytes at a time.