`ds-sim` Client, Stage 1

Benjamin Yee 45425108

April 2, 2023

Project Goal

`ds-sim` is designed to simulate the scheduling and execution of jobs in distributed systems. The system follows a client-server model, and provides a server system that simulates user job submissions and job executions. The task in this stage is to design and implement a client-side simulator for `ds-sim` that receives jobs one at a time, then schedules them for the server based on a pre-defined scheduling policy.

System Overview

`ds-sim` is designed to be language independent; however, the Client must follow a set communication protocol with a predefined vocabulary. Once the client has established a connection with the server and has received job and server information, it will schedule the jobs based on a *Largest-Round-Robin* (LRR) scheme, where jobs are sent to the server of the largest type. A typical interaction follows as such:

1. The Client starts a session with the server and authenticates a user. The server responds with the first job to be scheduled
2. The Client queries the server for a list of available servers, and selects the largest server based on the number of cores
3. The Client schedules a job to the largest server type, and requests another job
4. The server sends another job, and steps 3 - 4 are repeated until the server has no more jobs left
5. The Client leaves the session and disconnects from the server

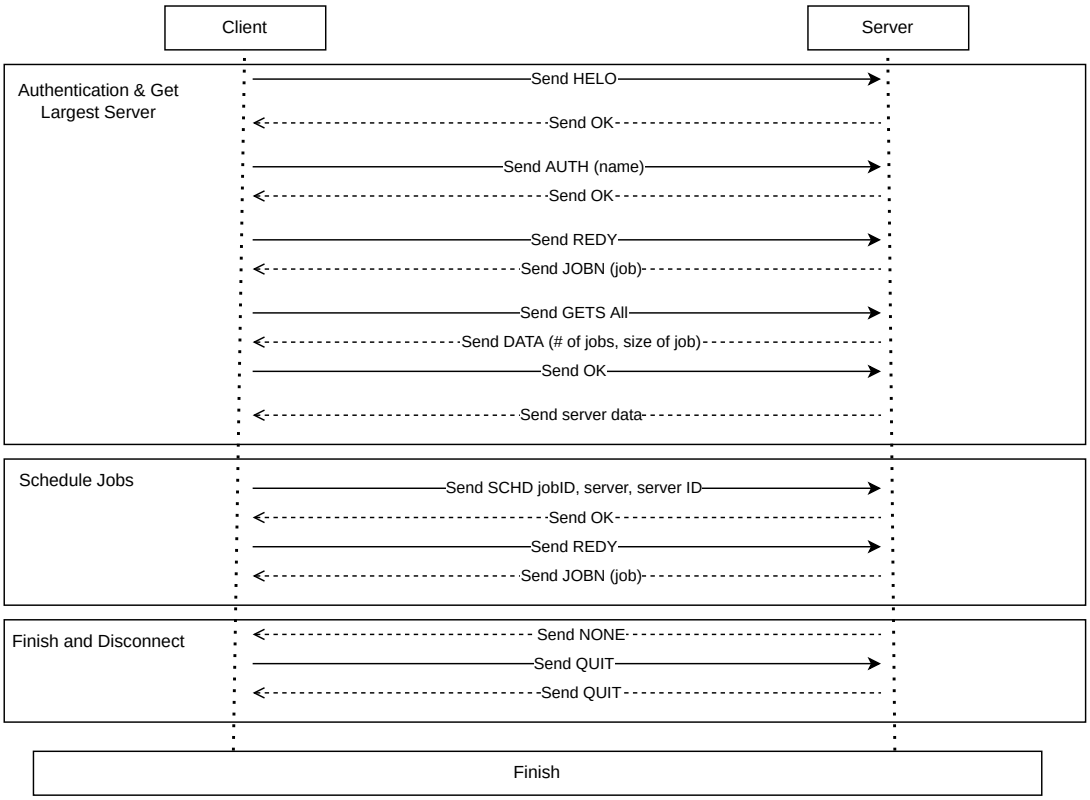

Figure 1: Sequence Diagram of a client-server interaction

Design & Implementation

As mentioned above, `ds-sim` is language independent, meaning that the Client can be developed in any language as long as the communication protocol is followed. As such, the client has been written and compiled in Java using the OpenJDK (17.0.6) Runtime Environment. Both the client and the `ds-sim` server will be run on an Ubuntu Linux (Kubuntu 22.10) system.

The Client consists of four main parts:

1. Initialisation, Handshake & Authentication

On launch, `ds-sim` opens a localhost socket on port 50000. The initialisation of the Client involves opening a reciprocal connection to the localhost socket, and creating input and output data streams in order to receive and send data to the server. Once a connection has been established, the client will initiate the session by sending a "HELO" code. Upon

receiving an "OK", the client will authenticate a user by using "AUTH (name)", and will send a "REDY" code when the authentication is accepted. The server will respond with a "JOBN" line, which represents a job to be scheduled.

2. Get Server Information

The *getLargest()* function, which queries the server for the largest available server type, was moved out of the main program to aid readability. The function sends a "GETS All" query to the server, which requests information on all servers regardless of state (which would usually be inactive at the start of the session). The first response from the server is in the format "DATA x y", where x denotes the number of servers. This value is used to create a String array that will store the identifiers of the servers listed by the `ds-sim` server.

Once the Client has received the list of servers, it iterates through the created String array to find the server type with the largest number of cores. In the case that multiple server types have the same number of cores, the client will choose the first of those server types in the list. The client will then create a second array, *largeList*, that contains all instances of the chosen server type. The total number of servers in the *largeList* array is stored in the *totalLargest* variable.

3. Scheduling

In the *Largest-Round-Robin* scheduling policy, the Client schedules jobs to the largest servers in a round-robin fashion. In this method, the client will send a job to each server and loop back to the first server when all servers have received a job.

The Client maintains a counter variable, *jobCount* that represents the ID of a job in the queue; this is incremented by one after a job has been scheduled. The round-robin policy is acheived by taking the modulo of the *jobCount* value and the *totalLargest* value.

The scheduling command of "SCHD *job ID server type server ID*" is then sent to the `ds-sim` server, where *job ID* is represented by the variable *jobCount*, and *server type* and *server ID* are stored in the *largeList* array. If the schedule has been successfully sent, then the server will respond with an "OK" message. Upon receiving the "OK", the Client will respond with a "REDY" message to indicate that it is ready to receive another job to be scheduled.

The server will also send a "JCPL" message whenever a job has been completed; the Client responds to these messages with a "REDY" message.

This process repeats until the `ds-sim` server has no more jobs left to schedule, at which point it will send a "NONE" message instead of a job. This is achieved through putting the scheduling and job-receiving functions in a while loop that operates as long as the received message is not "NONE".

The Client handles received messages by using a *switch case* statement, within the

prepNext() function that is run after each job has been scheduled. This method was chosen as it allows easier programming of extra responses in future development.

4. Disconnecting

Once the Client has received a "NONE" signal from the `ds-sim` server, it will exit the scheduling/receiving loop and send a "QUIT" signal to the server. The program will then finish execution at this point.

Notes on Implementation

Libraries

The Client imports three java libraries:

- `java.io.*`
  - Allows the program to handle file input and output operations, used to process the input/output streams active during communication with the server
- `java.net.*`
  - Provides classes for networking implementations, used to connect to the `ds-sim` server socket
- `java.util.ArrayList`
  - Used to create dynamic arrays - more useful than fixed-length arrays when creating a list of the largest server type

Exceptions

Interfaces in the `java.net` package rely on protocol handlers which must be present, otherwise an *Exception* is thrown [1]. Thus, since all functions in the Client contain interactions with data either received or sent to the server, all functions have `try...catch` statements to handle exceptions.

Variables

Currently, all global values in the Client are stored as public variables at the root of the class. This means that those variables may be accessed by an external program at runtime. Future iterations of this project may change those variables to protected to restrict access.

References

GitHub repository:

- COMP3100 unit content:
  - https://github.com/BenHasNoBrain/MQ_Stuff/tree/master
- Assignment Stage 1 content:
  - https://github.com/BenHasNoBrain/MQ_Stuff/tree/master/Assignment%20Stage%201

[1] - Oracle, “Java platform, Standard Edition Java API Reference,” java.net (Java SE 13 & JDK 13), 13-Sep-2019. [Online].

Available: https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/net/package-summary.html. [Accessed: 02-Apr-2023].