1 Introduction

The COMP3100 Distributed Systems scheduling project involves re-implementing the client-side simulator for `ds-sim`[1], a distributed systems simulator designed to be language-independent. In stage 1 of the project, we were tasked with implementing the basic functionality required for the client to work, which included:

- Connecting to the server
- Handshaking with the server
- Recieving and scheduling jobs
- Correctly quitting and disconnecting from the server

To schedule jobs, we had to implement a job scheduler known as Largest-Round-Robin (LRR), "that sends each job to a server of the largest type in a round-robin fashion"[2]. If multiple server types have the same number of cores, we proceed with the first server type listed.
The client-side simulator was to be programmed in the Java programming language, although the language of choice was seemed to be negotiable as long as it was readable, and it was to be delivered on GitHub[3] by the due date on the 3rd of April 2023.

2 System Overview

2.1 Server

The server-side program is responsible for simulating a distributed system based on a configuration file provided to it via arguments. The server generates jobs to simulate requests, and these jobs are then sent to the client to be scheduled. Upon the completion of a simulated job, the server will send the client a completed job message. Upon the total completion of job scheduling and when all jobs are done, the server will send a final message of `NONE`.

2.2 Client

The client-side program is responsible for establishing and authorizing a connection with the server, receiving jobs from the server, and scheduling these jobs based on the available servers. The client's main function is to schedule the provided jobs in such a way that it can make the best use of available resources. For stage 1, the client is only required to have the LRR scheduler implemented. This scheduler sends each job to a server of the largest type in a round-robin fashion. For example, with 3 largest servers, the order would be: Largest 1, Largest 2, Largest 3, Largest 1, etc.

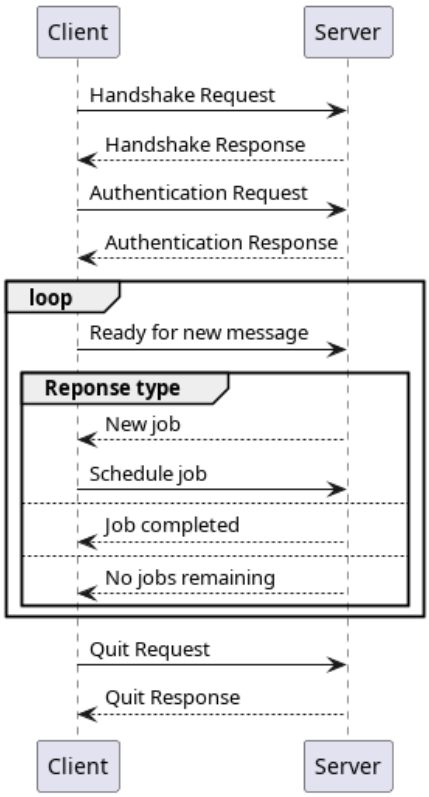

Figure 1: Sequence diagram of the system

3 Design

3.1 Philosophy

I designed the client-side simulator to be as simple and future-proof as possible. I wanted to be able to program a working solution fairly quickly, so I structured it around giving each method a specific, singular function it would perform. I did not hold myself too strictly to this, but enough that this approach would probably be clear to any reader. For future-proofing, I made sure that I would be able to easily update the code to account for different schedulers in the future. I based this on the information and short summary provided for the 2nd stage of the project.

3.2 Considerations

As this is an assignment, the implementation needs to be clear and quickly understandable for the purposes of marking. It also needed to be written in a way that I would be able to easily explain its functionality and structure weeks later. To achieve this, I wrote it in such a way that the primary steps the client goes through are clear in the main function, while the code doing the actual work is above in the Client class.
I also made the decision to omit launch arguments. While generally it's a good idea and it would fall in line with future-proofing, I didn't feel compelled to add it at the current time. This was primarily due to the current limited functionality of the program, but it also meant one less process to debug.

3.3 Functionality

When actually designing the system, I followed the design suggested by the provided pseudocode fairly closely. This resulted in a fairly straightforward design that worked almost straight away.

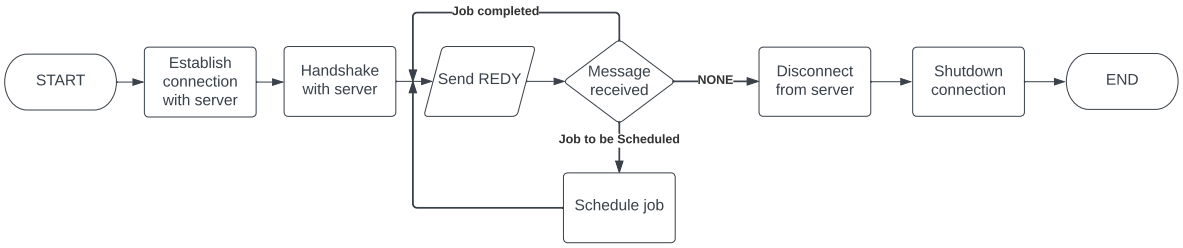

Figure 2: High level flowchart of Client program

The primary steps taken by the client are as follows:

- **Establish connection with server**: This is the first action taken by the client where the socket connection is established.
- **Handshake with server**: The handshake is performed, and a username is used to authorize the connection.
- **Send REDY**: The client sends a message to the server indicating that it is ready to receive a new message.
- **Message recieved**: A decision is made depending on the message received. If a new job message is received, it is scheduled using the LRR scheduler. A job completed message is currently ignored, and a `NONE` message starts the disconnection process.
- **Schedule job**: The job received is taken and scheduled using the LRR scheduler. Within this process, a request is sent to the server to return servers capable of handling the job to be scheduled.
- **Disconnect from server**: The client sends an exit message and waits for the response.
- **Shutdown connection**: The socket connection is closed.

4 Implementation

4.1 Technology

The client was written in Java with no addition non-default libraries, except for the ArrayList class which is part of the Java standard library. For quickly building and testing the program, I created a Makefile. By default, the Makefile compiles and runs the program, but I also provided options to perform these tasks individually and one for testing using the provided test files.

4.2 Data structures

- `DSClient`: The main class that the program was written and everything else interacts with
- `ServerJob`: This is a basic struct-like class mostly for the purpose of clean code used to store the data of a given job.
- `ServerListEntry`: Another struct-like class also for the purpose of clean code. But also used inconjunction with the `ArrayList` class to store the servers that the simulated server had returned.

4.3 Methods

The DSClient class contains several public, private and helper methods that implement the functionalities of the client.
The public methods are:

- `connect()`: Establishes a TCP/IP socket connection to the server.
- `connection_handshake()`: Performs the handshake with the server and authenticates the user.
- `schedule_job(String, AlgorthimType)`: Schedules a job given by a `JOBN` message and a given algorthim type enum.
- `disconnect()`: Sends the `EXIT` message to the server and waits for the reply.
- `shutdown()`: Closes the data output stream and TCP/IP socket.

The private methods are:

- `find_capable_servers(ServerJob)`: Sends a `GETS` command to the server to request information about available servers that can run the given job.
- `get_server(AlgorthimType)`: Gets the best fitting server based on the AlgorthimType enum argument parsed. Currently the only enum is `ALG_LRR`.
- `get_server_lrr()`: Finds the best fitting server using the LRR scheduler.

There are also a few helper methods which are:

- `send_and_wait(String)`: Takes the String argument parsed and sends it to the server. After which it waits for the response and returns the first line recieved.
- `error_mismatch(String, String)`: Basically a C-style macro for printing out if an message doesn't match what it's supposed to. Used in the handshake and disconnect methods.

References

[1] Y. C. Lee, "ds-sim: an open-source and language-independent distributed systems simulator." https://github.com/distsys-MQ/ds-sim, 2020.

[2] S. of Computing Macquarie University, "Comp3100 project stage 1 specification document." https://ilearn.mq.edu.au/mod/resource/view.php?id=7380298, 2023.

[3] L. McCarthy, "Comp3100 project repository." https://github.com/LeCarthy/comp3100Project.git, 2023.