Project report: Stage 1
COMP3100 Distributed Systems, S1, 2023
SID: 45156417, Name: Justin Ling

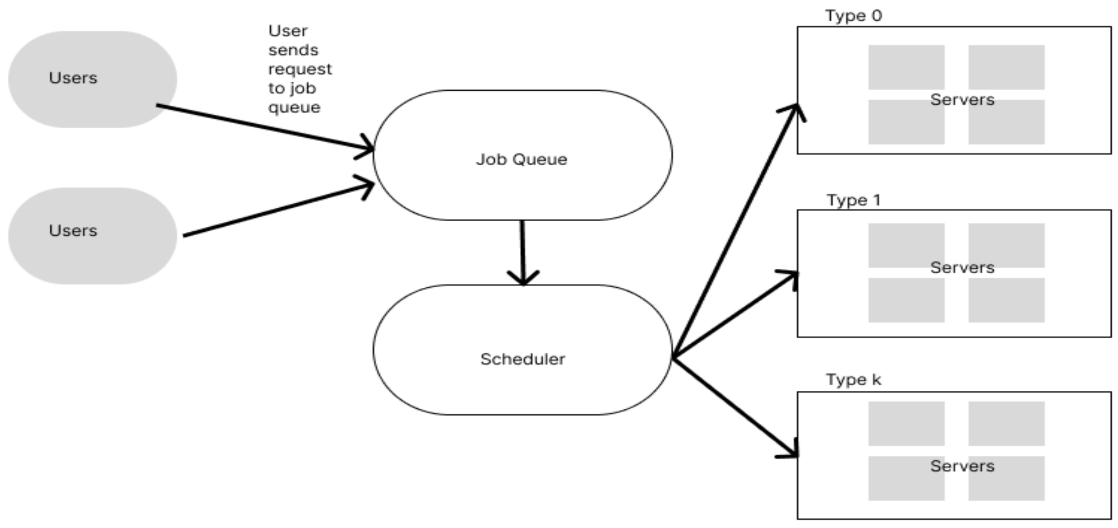
1 Introduction

Ds-sim is an open source distributed system designed to efficiently simulate job scheduling and execution between a server and a client [1]. The focus on this report is our implementation on a simple client-side simulator which includes basic scheduling and a simple job dispatcher. Simulating the client side implementation required creating a client-side simulator for a distributed system simulation (ds-sim) environment. The main goal is to develop a client-side scheduler that interacts with the server side to efficiently schedule and allocate resources for incoming jobs. Stage 1 specifically focuses on implementing a basic scheduler using the Largest Remaining Resource (LRR) scheduling algorithm and sending those jobs to the servers with the largest amount of CPU's available. This meant the implementation required key understanding in a client/server model, scheduling and dispatching.

2 System Overview

The system consists of two main components: the client-side simulator (scheduler job dispatcher) and the server-side simulator. The client-side simulator communicates with the server-side simulator to request and allocate resources for incoming jobs based on the LRR scheduling algorithm. A high-level workflow of the system is as follows:

- The client-side simulator establishes a connection with the server-side simulator and performs a handshake.
- It then requests job information and server capabilities from the server-side simulator.
- The client-side simulator identifies the largest server type based on the received data and schedules jobs accordingly.
- The client-side simulator keeps sending "REDY" messages to the server-side simulator based on the Largest Round Robin Algorithm to receive more jobs until a "NONE" message is received.
- Finally, the client-side simulator sends a "QUIT" message to terminate the connection and close the resources.



- Users - represented as individual entities that submit jobs that are added to the queue
- Job queue - Holds all incoming jobs that are waiting to be scheduled
- Scheduler - Responsible for assigning jobs to the appropriate server based on the largest round-robin algorithm
- Servers - Each server is grouped by a different type, these mean the resources available for each server and responsibility for each assigned job

3 Design

The design philosophy for our client-side simulator (ds-client) is to create a flexible, efficient, and easily adaptable system for job scheduling in a distributed environment. The primary focus is on the implementation of different scheduling policies and algorithms that can be easily integrated into the ds-client while maintaining compatibility with the ds-sim communication protocol.

Our design considerations include ensuring scalability to handle varying numbers of jobs and server resources, maintainability for future updates and improvements, and being extensible to accommodate new scheduling algorithms. Additionally, we take into account the constraints imposed by the ds-server, such as communication protocols, data structures, and the simulation model.

3.1 Design Philosophy

The main design philosophy for our client-side simulator is a develop an efficient and easily adaptable system that allows us to execute any amount of job scheduling from the server based on varying job sizes, diverse server resources, and workload demands. Furthermore, our implementation kept modularity and extensibility in mind which was to separate our code into separate modular components to ensure most of the functionality can operate independently as well as adding new features in the future can be easily incorporated.

3.2 Considerations

When designing the client-simulator, several important considerations were taken into account to ensure compatibility, usability, and performance. Firstly the choice of language to develop the ds-client was Java as it provided extensive libraries, modularity, and support from the community.

Scalability was also a major consideration when designing the implementation as the client needed to be able to accept a range of scenarios. To address this, the design had to ensure the right data structures and algorithms were utilized so that the system can run efficiently without using more resources overtime.

Finally, a thorough consideration between the ds-client and ds-server required defining clear message formats and clear error handling to ensure any potential errors or failures can be mitigated or found easily to improve debugging and minimise disruptions.

3.3 Constraints

The main constraints and possible improvements of the ds-client implementation is:

- Limited server resources - Lack of unique CPU cores, memory, and storage sizes which ultimately influenced how the scheduling and dispatching of the algorithms operated.
- Job priority - The implemented algorithms did not account for any time-sensitive scheduling or priority jobs based on any demand for urgency and duration.
- Modular code - The implementation of the ds-client did not separate the algorithms into independent methods which would have helped improve testing and performance potentially.

3.4 Components

The handshake component is the initial starting point of the ds-client. This involved establishing and maintaining a connection between the client and server which was established with a socket. It handled the initial communication protocol which followed the ds-sim communication protocol guidelines.

The retrieval component's main functionality was to get all the capable server types and their resources. This component sent a query to the server and the server responded by providing a list of relevant servers such as the server name, number of cores, memory, disk space, and server type. This information was important to allow us to start allocating jobs to each server.

Lastly, the termination component was responsible for the closing of the connection between the client and the server. This was done once the client sent the message ”QUIT” to the server and the server accepts this message, the client will then close the socket, input, and output streams

4 Implementation

The client-side simulator is implemented in Java with version 11.0.18, utilizing standard libraries for socket communication, input/output streams, and java's utility library. The simulator is divided into several modular functions, each responsible for a specific task:

4.1 Send function

The send function served the purpose of sending messages from the client to the server.

- The function took a single input which was the message from the client to the server
- The message string is then converted to a byte using the getBytes() method. This was because java's output stream required messages to be sent as a byte object.
- The write method was called to send the message to the server and then the flush was used to ensure the data is sent.
- To debug and understand the communication of the client and server a println was used with the ”C” in front to identify that the message sent was from the ”Client”.
- If any exceptions happened during the process, they would be caught and displayed in the console.

4.2 Receive function

receive(): receives messages from the server and processes them in the client.

- din.readLine is the bufferedReader object which allowed the line to be read from the output of the server. Which was stored in a variable called messages.
- The received messaged is printed out to the console starting with ”S” to identify that it was a message from the ”Server”.
- The function then is returned back to the static message variable allowing it to be used in the main function.
- If any exceptions were caught it would be displayed in the console as ”Exception from RECEIVE” and then return null indicating that no valid message was received.

4.3 Main function

The implementation of the main method was to execute all the messages sent between the client and server. It included establishing the initial handshake, getting the largest servers, the LRR algorithm, and finally closing the connection.

- First it establishes the socket connection at the IP address 127.0.0.1 with port 50000 and initializes the IO streams.
- Next it setups the initial handshake between the client and server by sending ”HELO” and then ”AUTH” with the system name and signals a ”REDY” to the server.
- Next it establishes a parameter to query the ”GETS Capable” to return the servers based on cores, memory, and disk.
- The servers are then run in a for loop which is to identify all the largest servers and store them in an ArrayList called serverId's. This is to keep track of all the capable servers to send the jobs to.

- Next the Largest Round Robin algorithm is set by a loop. For each job received that equals to ”JOBN” The loop schedules the job to the next available server in the list of the largest servers from the ArrayList. The algorithm utilised a currentIndex variable which kept track of which servers were in the list being used by the current job. So by incrementing the index it moves to the next server. The line with the modulo ((currentIndex = currentIndex + 1) mod serverIds.size()) is used to back to the beginning of the list when it reaches its end of it. This way servers in the lists all get utilised and cycle through the list in a round-robin process.
- After all the jobs are scheduled until a ”NONE” is received the client sends a message to the server saying ”QUIT” to close the connection and then finally closes the IO streams and socket.

5 References

- Link to Justin’s Github — https://github.com/Justin-zi/COMP3100
- [1]Distributed System Documentation ”distsys-MQ” — https://github.com/distsys-MQ/ds-sim