Client-Server Cloud based simulator

By:
Bruce Liu (45958114)
Aidan Burdett(45617864)
Daniel Nguyen (43653812)

Introduction:

Distributed systems come in a wide variety of sizes and magnitudes, which can range from an individual workstation computer with multiple CPUs with multiple cores, a cluster of compute nodes (such as blade servers working together in a group) to multiple data centres (with multiple server racks) such as Amazon Web Services and Google data centres. To effectively utilize the computing resources, we must schedule tasks to be performed as efficiently as possible, hence the purpose of this project is to address that task. In this project we use the client-server model and a simulated distributed system with user specified configurations is paired up with the client-side simulator which contain scheduling policies and algorithms. Stage 1 of this project focuses on the design and implementation of the client side of the simulator (known as ds-simulator) that addresses the job scheduling and job dispatching based on given configuration(s), which was done using Java programming language and GitHub for collaboration.

System Overview:

The project is designed to simulate realistic job scheduling and execution for distributed systems such as computer clusters and data centres (represented as Figure 1) with repeatable results. The ds-sim consists of two main parts:

- ds-client
- ds-server

Our implementation of ds-client works by

1. handshaking with the server and verifies authorization based on username
2. goes through the list received by the server and store it in a separate class
3. determining the largest server (based on core count)
4. allocating to largest server based on given configuration(s)

The client side schedules the jobs, whereas the server side simulates the job submissions and job executions to servers. The client side is flexible in that it can accommodate for a variety of scheduling policies and algorithms given that it communicates with the server side on the same protocol(s). On a high level view ds-sim follow 3 steps:

1. ds-server provides job and sends to ds-client
2. ds-client decides how to schedule the jobs and sends to ds-server

3. ds-server executes the jobs based on the ds-client's choices of scheduling.

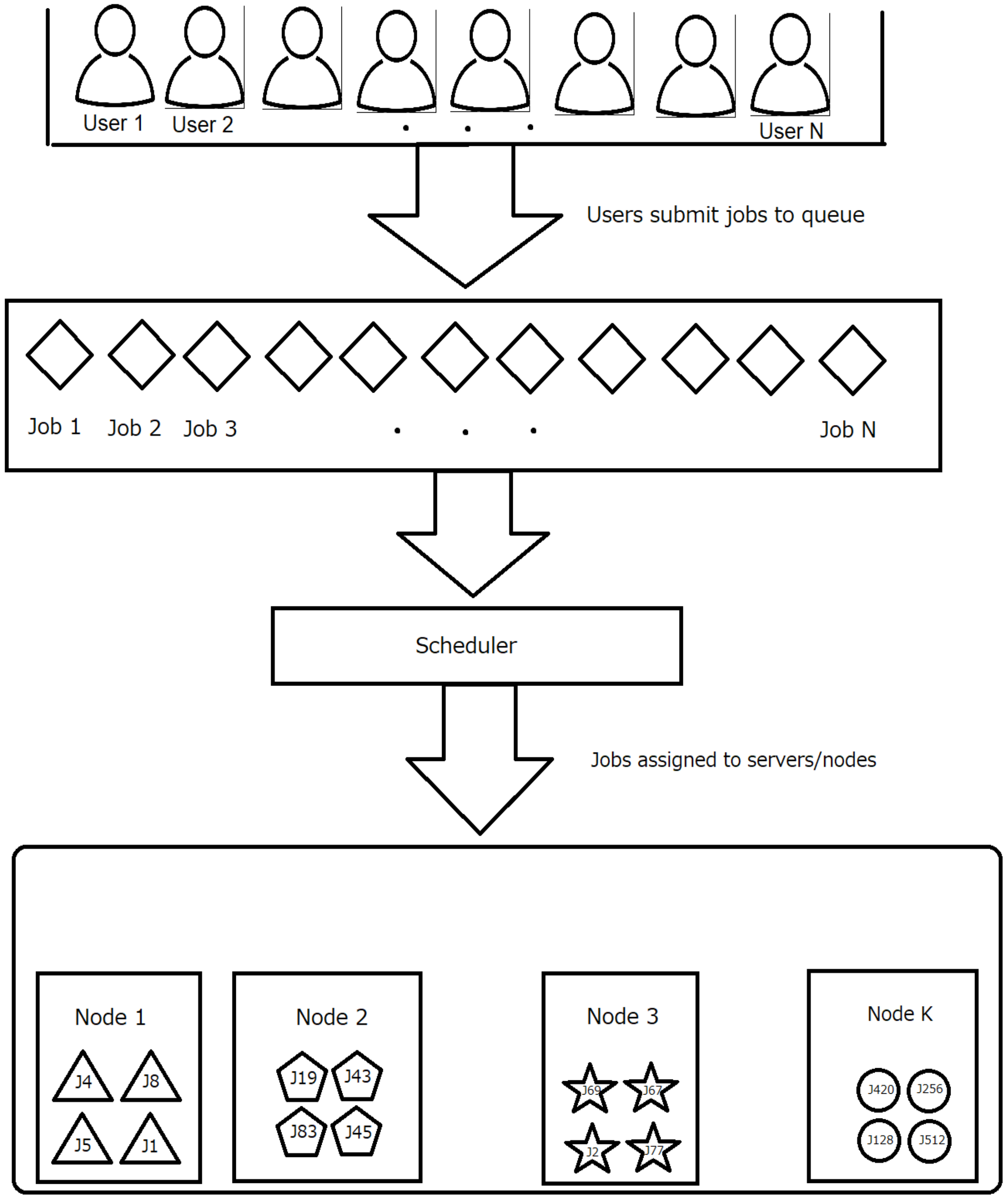

Figure 1: How jobs are scheduled in a typical distributed system.

Design:

Regarding the design philosophy of the project, we prioritised a focus on readability and simplicity in order to not complicate the process of designing and understanding the project's various components. We made sure to allow for modularity in case any improvements needed to be made further in the process of constructing the program.

The above considerations led to some constraints that we had to impose on the project, namely; The project needed to be simplistic which restricted our ability to implement complex methods, and it needed to be modular which resulted in a fairly loosely coupled system which is ultimately worse for handling large amounts of data as a result of more varying distinct components. These constraints proved not to be very troublesome for our planned requirements as we did not need any alterations to the scope of the project and it remained simplistic.

The Client-Server Cloud Based Simulator is made up of several components with differing functionalities;

1. **The Client** - handles all operations that communicate with the server. Communicates to the server through a socket. It begins by performing a handshake operation with the server and then begins taking in jobs to schedule. It inspects the available servers' specifications then determines the most effective server for a given job based on its size and schedules the job.
2. **The Storage Class** - provides a data type that stores relevant information about each server for the client to determine which server to best handle a given job by comparing the attributes' contents.
3. **The Servers** - provide simulated 'hardware' that processes the jobs that are given to each one. Statistics are given about each job performed in order to create a report of the performance of a server on a specified job.

Throughout the design of our project, we focused on ensuring that these three components were able to interact smoothly and without needless complexity. This goal was achieved through our implementation of the Client.java class and its interaction with the servers through a middle communication class, the Storage.java class. The ability for the Client to interact with the server in this way allows for modularity and breaks down the system into distinct separate parts which assists in understanding any potential errors that may have arisen.

By designing the project with consideration of the aforementioned principles, we were able to construct a working client-server cloud based simulator with relative ease. Its operation is clear to understand and its modularity allowed for easy expansion of its functionality for any potential future additions.

Implementation:

In terms of general implementation of the program, foundational technologies utilised included but not limited to the following;

1. **Ubuntu version 20.04 -** Allows for the installation of a Linux based operating system environment, to be used in tandem with VirtualBox in order to develop a separate Linux environment for windows based computers. Must be done in order to solve compatibility issues with ds-sim code.
2. **VirtualBox version 6.1.18 -** Allows the generation of a virtual machine on current computers in order to allow for a new development environment with its own individual resources that will not impact the original PC environment in case of errors.
3. **Visual Studio Code version 1.55.2 -** Allows for an idle development environment that is visually friendly. This is considered to be an optional piece of technology but is important for indentation and overall structuring of written code. Visual Studio additionally provides options for multiple bash terminals that can be used for testing all in a single location.
4. **Java version 16 -** Allows the usage and creation of Java based code files, essential in order to run Java based programs and applications.

In terms of a more code specific perspective, specific libraries included in the program is as listed below;

1. **Java.net -** Foster general connection between the server and client on the same socket in order to allow sharing of information.
2. **Java.io -** Required in order to develop the read and write functionality for the program in order to foster communication between the server and client.
3. **Java.util -** Mainly used for the implementation of arraylist to allow dynamic allocation of storage for server and job scheduling information

In direct relation to Java.util, the most common data structure utilised in our rendition of a client-server simulation is the ArrayList which is closely related to the array. ArrayLists are mainly used to store data in regards to both the server information and job information received from the server. The general array is also used within the methods that separate the received strings from the server, it helps to store the different segments of said string upon separation into different segments. There is also the usage of regex techniques in that the string “\\s+” is included as a means of separating a long string into smaller ones based on the detection of whitespaces in a sequence which is comparatively different from “\\s” which only detects a singular whitespace. Apart from the above mentioned there is also the addition of a Storage class which is written on a separate file which directly acts in accordance with object oriented programming practises and is called upon in the Client file as a custom class storage object.

As a basic rundown of the entire program, the code has four main components and holds five different methods.

Methods:

1. **Initialization -** Initializes the static strings for storage of commands and creates the socket along with a printwriter and bufferedreader. Also creates separate string, arraylist and object variables in order to store future information from the server.
2. **Handshake -** Starts the handshake process beginning from HELO to the point of sending REDY to allow communication between the client and server, uses write method mostly.
3. **Information sorting -** Receives Job information from the server and stores it, utilises the three methods of Separate, getLargest and CurJobID in order to develop and change the received strings into usable information for the rest of the program. Generates a Storage type object to store the largest server and a string in order to store the jobID of each job dynamically.
4. **Job allocation and quitting -** Indicates the conditional steps to take once receiving a certain command including that of JOBN, JOBP and OK. Uses the function allToLargest in order to automatically schedule all jobs to the largest server. Quits program and closes all connections once receiving the QUIT message.

Functions:

1. **Separate -** Written by both Bruce and Aidan, receives an arraylist of strings and goes through each string on a single for loop and splits the strings into smaller strings using \\s+. Stores smaller string into a temporary string and within the for loop, allocates the smaller segments into a Storage object to be later stored into an arraylist of type Storage, returns the arraylist of type Storage.
2. **getLargest -** Written by Daniel and Aidan, simply accepts an arraylist of type Storage and returns a single type Storage class that contains the largest number of cores, if there are multiple largest cores of the same value, returns the first one that is found in the loop.
3. **CurJobID -** Written by Daniel, uses the idea from Separate method and separates the received jobs from the server into smaller segments and simply calls upon the second segment to be returned.
4. **allToLargest -** Written by Bruce, takes a single Storage object and String in order to schedule the JobID in the String to the server stored in the Storage object.
5. **Write -** Written by Aidan, simply replaces the need to continuously type in pw.println and pw.flush() and contains them into a single method.

Each method has been attributed to the overall design and development of the program, the Separate method allowed for an alternative method to xml parsing and directly inspired the idea in the development of CurJobID. getLargest determines the server for jobs to be scheduled to and hence is a vital part of the code while the allToLargest and Write improves overall readability and appearance.

References:

- **Github -** https://github.com/aidanburdett/COMP3100-Group-48