**Distributed Systems Job Scheduler Using Largest Available Server**

Luke Glover - #45965587

Jarrod Adair - #45432260

Frederick Jack – #45511020

**Introduction**

Introduction (½ page): What this project (focusing on Stage 1) is about, including the goal of the project and Stage 1.

Distributed systems are comprised of components that can be present in different locations but still function as one; coming in many different types and sizes, ranging from single machines with multiple cores to data center with millions of servers spread globally, making distributed systems one of the indispensable concepts of the modern computing age. When dealing with these multiple distributed components acting towards a similar goal under one or more constraints, Scheduling is considered the key technique for ensuring the efficient use of computer resources.

This distributed systems project, focusing on task 1, is about building a “Vanilla” client in the Java language that is compatible with the ds-sim server simulator provided. This required us to build a job scheduler client that could schedule jobs via discrete sequences of commands sent over a communication layer, with the goal to dispatch every job to the largest server available in each simulation.

In a distributed system job allocation is very important for the efficient use of resources. Scheduling is a decision-making process that typically uses an optimizing algorithm for dispatching of jobs. This project is implementing a client-side job scheduler working with a distributed systems server simulator. The task of Stage 1 of the project is to schedule and dispatch every job to the largest server available in each simulation. The client is connected through a socket, so it is language independent, but this implementation is written in Java. The goal of this stage of the project is to successfully run and scheduled all available jobs without failures.

**System Overview**

System overview (½ page): high-level description of the system (both client-side simulator and server-side simulator with the focus being your client-side simulator), preferably, with a figure (your own, not one in ds-sim User Guide) showing the workflow/working of the system.

At a high level, the system is started by having the pre-compiled server listen for a connection and the client initialising that connection with the server thought java sockets provided by a library. This socket acts as the medium that allows the Client and Server to connect via TCP protocol (Transfer Control Protocol) and send messages about job information back and forth. The Client’s core functionality is essentially a large loop that functions as a state machine. Firstly, the Client initiates a connection the Server via the handshaking state code and directly proceeds into authentication. The Server then reads the specified configuration file and writes a ds-system.xml file which contains the servers that are available for the Client to use. The Client will read the ds-system.xml file to find out that information. After initialisation, the event loop sends a ready message which the server replies with a single jobs details for the client to submit for scheduling. For state 1, the Client then sends a command to the Server for it to process the job with the largest server it has available and once the Server has processed a job, it sends a completion message and repeats this process until there are no new jobs and the Server tells the client it has no more jobs. Finally, the Client then sends a quit command to the Server and closes the connection. The output the scheduling is stored in a ds-jobs.xml file and it contains information about each job’s scheduling as well as other information.

The process is started by the Client initiating a connection with the Server through a socket. The socket is the middleware that connects the Client and Server and allows them to send messages to each other. A Java socket uses TCP (Transfer Control Protocol) to create a connection. The steps are that the Server sends a single job with details to the client for scheduling. The Client then sends a command to the Server for it to process the job with the largest server it has available. Once the Server has processed a job it sends a completion message, and this process repeats until there are no new jobs and the Server tells the client it has no more jobs. The Client then sends a quit command to the Server and closes the connection. The output the scheduling is stored in a ds-jobs.xml file and it contains information about each job process.

![Diagram

Description automatically generated]()

As showed in the figure above the distributed system has six components.

socket – is the process that connects the client and server through a local host on port 50000.

server – also known as ds-server, simulates a cloud server and is written in C. The socket allows the client-server connection to exist without having the same language.

configuration file - contain the information about jobs such as the submit time, job ID, estimated run time and the resource requirements needed to process.

ds-system.xml file - contains the information about the servers available: how many of a particular server the bootup time the cost and the server resource specs.

client – program written to receive messages from server and to act as a job scheduler.

ds-jobs.xml file – contains the output of processed job after the connection is closed.

**Design**

Design (1 page): design philosophy, considerations and constraints, functionalities

of each simulator component focusing on the client-side simulator.

Considerations and constraints:

* Java lang client to C lang server relationship. We had to keep in consideration the differences in these such as new line /n characters in messages
* Possible constraint?? Timing between client and server. Server is a lot faster so we had to add timing delays for tests

Functionalities of each simulator component focusing on the client-side simulator:

* Main class with loop where all functionality stems from.
  + Within loop is case checking to set the client’s state.
* Split every part of functionality into different classes to have clearly defined components of the client. Good OOP practice.
* Connection class handles all possible socket interactions between the client and server. Handles sending and receiving data in formats that are readable by both systems.
* SystemInfomation class stores all information about data that may be used globally within the client as well as server information. It partially works in conjunction with the XMLParser to add the parsed server data into actual useable collections for the server to use when scheduling jobs as well as determining server related functions such as highest core count.
* Server class is used as a data structure to hold all possible information about an individual server. Client will parse server data and create multiple instances of this server class to keep track of each server that is available to it.
* Job class Is used as a data structure similar to server and holds all possible information about an individual job.
* Action class holds some of the backbone logic for our client, outlining actions and setting up the ability to respond to server data. IDK write more of this one!!
* XMLParser class handles all parsing interactions with the xml files created by the server.
* ProtocolState, ServerState, ActionIntent are all enumeration classes that assist in the client’s state system. These hold all the unique states that the clients components can be in at any one time.
* UnrecognisedCommandException class handles any exceptions that may occur during the runtime of our client.

Our DS-Sim client was modelled after a state machine. The behavior of the client is based upon the so-called state of the larger system, and certain actions or events trigger the client to switch to another state. These states are loosely based upon the command categories given in Appendix B of the DS-Sim User Guide [1], with some modification for technical reasons.

This design was chosen as it provides several benefits. Firstly, it is easier to handle cases where the same command has different meanings, depending upon the context. For example, the “OK” message is sent by the server in response to “HELO”, “AUTH” and “SCHD”, and what the desired behavior of the client in response varies in each case. Secondly, it prevents classes becoming unmanageably large by segmenting code into discrete parts that are each responsible for a small part of the simulation. Thirdly, the design allows for the implementation of alternate scheduling algorithms without any major modification to the main method or any other classes, which is particularly useful for stage 2 of the project.

Our implementation of a state machine is based upon the strategy design pattern, which allows the behavior of a program to change at runtime [2]. The behavior of the client in each state is defined in a class that implements the *ProtocolHandler* interface, which is called from the event loop in the client main method.

The protocol handler classes can queue certain actions such as sending messages to the server and changing state by returning an instance ofthe *Action* class to main, which instructs the main method to carry out the behavior described by the Action object. This project structure leads to readable and meaningful code, which was an important objective of the project, as it will have to be revisited later by team members for stage 2 of the project.

Another important design consideration was the use of object-oriented programming (OOP) practices. We made use of Java’s many OOP tools to build an abstract representation of the simulation and its components. This abstraction simplified development of the protocol implementation and scheduling algorithm: rather than constructing strings to send to DS-Sim server, higher level code instead passes Java objects to lower-level methods, which translate these objects into strings before transmission. A brief overview of the major classes are as follows:

* The *SystemInformation* class is a static class that is callable from anywhere in the project. It provides read only access to a variety of system-wide variables, such as the remote address of DS-Sim server, the command line arguments that the client was started with, and a list of all job execution servers available in the simulation.
* The *Server* class represents a server in the simulation, i.e., the servers that execute jobs, not the DS-Sim server with which the client communications. It is a data structure that stores information about each a server, such as an ID and a type.
* The *Job* class is like the *Server* class but represents a schedulable job in the simulation rather than a server.
* The *Action* class (introduced above) provides instructions that are executed by the main event loop. This class allows more complex communication from the protocol handler classes to the main loop than any primitive datatype. For example, the Action object Action(ActionIntent.SEND\_MESSAGE, "HELO")returned from an protocol handler causes the message “HELO” to be transmitted to the DS-Sim server.

**Implementation**

Implementation (2 pages): brief description of any implementation specific information including technologies, techniques, software libraries and data structures used. How each of components/functions of your simulator is implemented including who oversees which function(s) and how they have led the design and development.

who oversees which function(s) and how they have led the design and development:

* Initial development of most systems within the client, design and implementation of Job, server, systemInfomation classes, general clean-up of code: Jarrod
* Design and implementation of state structure and protocols of client, general clean-up of code: Luke
* XML parsing: Fred

GitHub Repo: <https://github.com/Luke-Glover/DS-Sim-Client.git>

[1] Y. Lee, Y. Kim, and J. King, ‘ds-sim: A Distributed Systems Simulator User Guide’. [Online]. Available: https://github.com/distsys-MQ/ds-sim.

[2] E. Gamma, *Design patterns: elements of reusable object-oriented software*, 37th printing. Reading, Mass: Addison-Wesley, 1995.

References including project git repository/wiki, e.g., GitHub and Bitbucket.

\* The numbers of pages for each section are also a suggestion.