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

Luke Glover - #45965587

Jarrod Adair - #45432260

Frederick Jack – #45511020

**Introduction**

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.

**System Overview**

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 (Transmission 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.

![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**

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**

Our client was implemented in Java version 15, the latest version of Java when the project was commenced. Specifically, it was developed using the Adopt JDK, an open source implementation of the Java development kit. The team members used a combination of JetBrains’ IntelliJ IDEA and Microsoft’s Visual Studio Code to develop the client. The client was tested on Ubuntu 20.04 LTS in a virtual machine.

With Java 15, we could use several language features not available on previous versions of Java. A notable example is the enhanced switch statement used throughout the project, available since Java 14. We made extensive use of the Java standard library, including for TCP socket connections, XML parsing and I/O handling. No other third-party libraries were included in the project.

Our client implementation includes a *ProtocolHandler* interface, of which there are several implementing classes. This interface is key to the strategy design pattern discussed above. The client main method passes all messages received from the DS-Sim server as strings to the *ProtocolHandler* via the *onReceiveMessage* method. The class is then able to process this message in any way, and return an *Action* object describing the desired response. Depending on the protocol state, the main method passes messages to a different implementation of *ProtocolHandler,* for example *HandshakeProtocolHandler* implements only the first two messages, after which the client begins processing messages in the *AuthenticationProtocolHandler*.

The Job and Server class components of the client were both designed in a similar way as to hold data about jobs and servers available respectively; while they don’t use any specific technologies or software libraries they were designed around data and their data structure hopefully shows that. These classes use a data structure mix of data storage as well as state storage for the intended use for every instance of a job or a server to be a new object of the class. These classes contain a constructor that is used when creating a new instance object which forwards the data into local variables to be stored in that specific instance of the class, allowing for multiple jobs or servers to exist at one time. Additionally, we added the ability to store the current job and server state inside each instance which further improved the compatibility with our general state machine design. These two classes fit into the client as a whole by giving it the functionality to remember which is essential for it to be able to perform scheduling tasks. These components of the client were designed and initially implemented by Jarrod with multiple small refactors along with Luke refactoring general code and shaping it into the Action state system we had developed.

The SystemInfomation class is loosely coupled with the server class as well as a general loose coupling with the overall client, acting as an umbrella global class to store globally accessible server objects as well as all basic client-server connection arguments. When designing this class, we chose to use the singleton design pattern as it felt the best suited for the behaviour expected. Inside SystemInfomation, we imported and used the ArrayList data structure of type server to store a multidimensional array of server objects with an accompanying addServer function and mostCores function to add servers and return the largest server in relation to their core counts, respectively. This class was originally designed and developed by Jarrod with help from Fred and with later refactoring the singleton class into a static class which still performed the same but with less code by Luke.

The Connection class is one of the most important and essential classes of the client with it solely in charge of the socket connection between the client and the server. This class uses external socket libraries that provide the backbone code for possible connections. Additionally, the connection class uses a combination of stream reader and writer data structures to parse byte streams into complete and readable strings for the client to process. The connection class includes multiple functions to perform discrete actions such as writing to and reading from the server, using stream writers/readers initialised within the constructor which is called at the beginning of the client start-up perdure. The initial creation of the connection class was from the whole team partially with Luke extrapolating the design into its current form with a few extra checks and refactoring.

The XMLParser class was created to parse the ds-system.xml file which was a necessary option we took for our client’s design, which uses org.w3c.dom, Java’s built in Document Object Model (DOM) and general File IO libraries. The input is an XML file written by the Server after it connects to the Client. The information contained in the file is all the server information and attributes available in the specified configuration file. The output after the method completes is the parsed server attributes stored in the correct data type and placed in an array list for use by the Client when scheduling the job. The steps used to parse the document are:

* Import the applicable packages: org.w3c.dom.\*, javax.xml.parsers.\*, and java.io\*
* Create a Document Builder
* Create a document from the file
* From the new XPath object created it can read the data and use the expressions to gather what you need from the document
* Retrieve a node by a specific tag name, in this case it is the “server” tag in this instance it is “//system/servers/server[@type]”
* Examine key-value pairs that contain the server attributes

The values read from the XML file are read as string data but need to be parsed again to the appropriate data type. The values are then passed into and stored in an array list of servers. This design was created and implemented by Fred, with small refactors by the rest of the team though out its development.

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.