# Stage 1: Simple Job Dispatcher

**Project Title**

Automated Resource Allocation for a Distributed System

**Group Members**


| | | |
|---|---|---|
| Bradley Kenny **45209723** | John Kim **45163782** | Mark Smith **45176655** |


**Introduction**

The overall aim of the project is to implement a client that is able to understand information from a host of servers and send jobs to a particular server depending on the required and available resources. Stage 1 of this project was focused on implementing a vanilla implementation that could dispatch jobs to a server. By design, we have been tasked with sending each job to the largest possible server on the cluster. With the server being given, we are tasked with designing a simple job dispatcher client that is capable of interacting with the server to allocate jobs.

**System Overview**

Design of the client required a detailed study of how the server had been implemented. We needed to completely understand the requirements of the server and how we should interact with it. The server has been set-up to work on the localhost, `127.0.0.1`, and through port `8096`.

Building up on our previous computing units and the importance of "good" coding practices, we wanted to create a design philosophy around forming methods in parts. Knowing that this project builds up on itself, we wanted a clean environment from the start. To support this philosophy and to help all 3 team members, we heavily encouraged to make comments so we know what the function is designed to do and how it was created. Furthermore, each method name was self-explanatory and if there was any doubt, we included a description of what it should do.

One of the main challenges we've faced was trying to reverse engineer the server so that our client program could interact with it as intended. As a team, we were able to breakdown the functionalities of the server and how our Java program could be tailored to match the C-written program. Given the server source code visibility restriction, we had to figure out the methodologies via trial and error which proved to be a big time sink. Using the provided documents as a reference and results from constant tests, we were able to produce an output that matched the expected output. As seen in *figure 1*, we recreated the protocol in a diagram that was more understandable to us. This allowed us to understand how to implement the protocol when creating our client program.

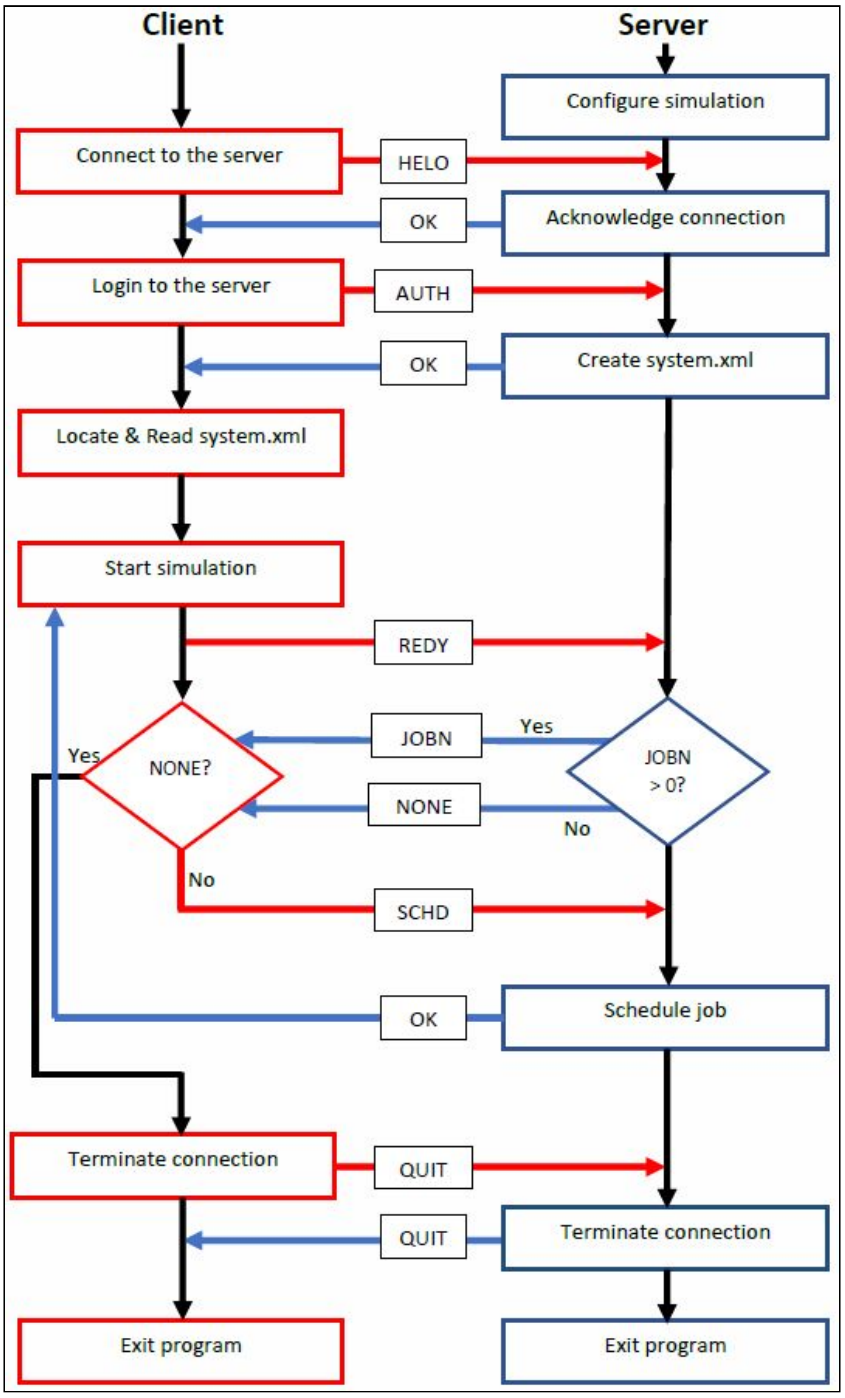


*Figure 1: Vanilla job dispatcher flowchart.*

Due to our usage of Java, we had to learn how to get it to cooperate with a program written in C. This was mostly troublesome to begin with as we couldn't use constructs that would usually work in a Java implementation. However, we were able to quickly learn how to adapt our program to work with the C implementation and provide it some input that it can understand.

**System Architecture**

Our client is split into a number of methods and classes. We have a main class containing the methods and properties necessary to communicate with the server, and another that is used to hold information relating to the XML server data.

We decided to split it in this way to ensure that our code was logically broken up and we were able to distribute the workload by each method. Furthermore, it made the code more readable, and understandable. It was important for us to ensure a certain level of readability as there are multiple people working on the project and we did not want to spend time explaining everything that is happening in the file. Comments throughout the methods assisted our internal team and any potential external viewers whom may want to see the source code and understand its function.

Our implementation involved the use of two applications that communicated through a common port, `8096`. This allowed them to communicate messages using the provided protocol. Through using our architecture, we were able to ensure that messages were being sent through the transfer agent, and finally received by the server agent on the other end, as well as vice versa. It was important to ensure we were compliant with the protocol.

To do this, we needed to learn how to implement socket programming. This is used to allow two applications to communicate with one another [1]. To get this to work, we used Java APIs that did the low-level stuff for us [3]. This ensured that we were able to set up the connection and transfer information between the applications without having to create much of the work needed for the socket connection ourselves. Furthermore, we used other APIs to allow us to read and write information using the socket. With this, we then implemented our client so that it would be able to understand and react to anything sent from the server.

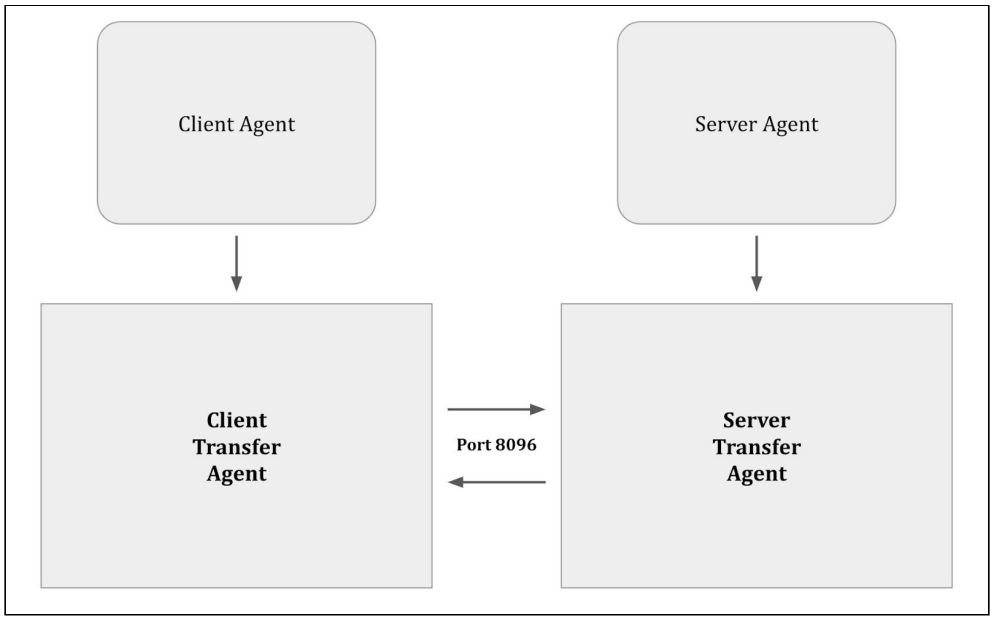


*Figure 2: Architecture of implementation.*

Our client contained a number of methods responsible for sending, receiving, parsing XML, among other things. It was important that each task the client had to complete was its own method. This allowed us to redo this task later, if needed.

During the process, we learnt that the architecture is based on the SMTP model [2]. As such, we were able to research into this protocol to obtain a better understanding of how to implement our client.

## Implementation Details

Our implementation is built using the Java programming language. Having the most experience in developing in Java throughout the university degree, it was thought to be the best fit for this project. However, since the provided example codes and the working server was written in C, our project faced various compatibility issues; one such example was the server passing data in bytes where Java natively understands primitives such as Strings.

Furthermore, we imported a number of modules that enabled us to use features of the programming language to complete the tasks required. One example being the XML Parser. An XML file was produced by the server and the client had to locate the file and read the attributed information from various tags. Those attributes were then stored within the Java function assigned to appropriate data types. One of these attributes is required to determine which server is the largest to satisfy the project "`allToLargest`" policy requirement. This function is then used to be able to determine which server we should be dispatching the jobs to. Given that this was specified to be the server with the largest number of CPU cores, the functions locates that server and dispatches the jobs.

In order to use the client, you need to have the server program running using the desired config files. This program should create an open connection on the specified port. From here, you run the **compiled** Java file using the command `Java Client`. The client will then simulate a job dispatcher. We have options in our code to be able to print everything the client sends and receives but commented these out for the submission as the demonstration client did not print this information.

We distributed the workload so that everyone would be able to contribute and no one person was developing the entire project. Brad was responsible for the socket connection between the client and the server, sending and receiving messages. Mark was responsible for the XML parser and job dispatcher sequence between the client and server. Finally, John was responsible for general assisting and testing the program as well as leading the document development.

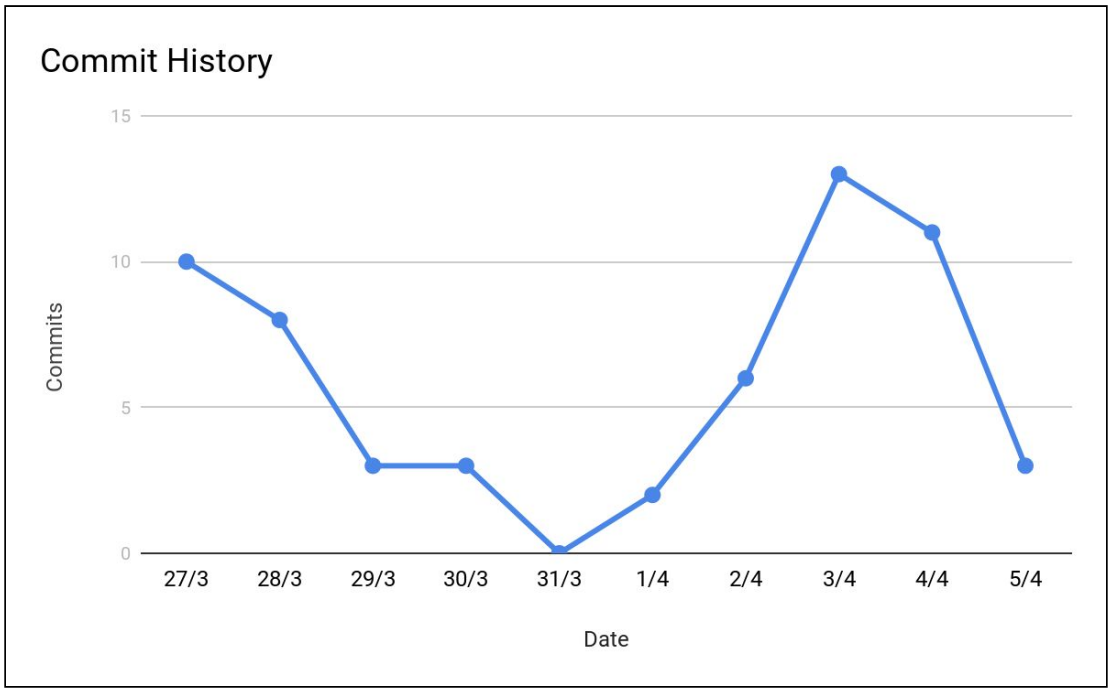


*Figure 3: Line graph displaying commit history for stage 1.*

As seen in *figure 3*, our work was mostly consistent from the date given until the due date. Dates with lower commits were usually due to struggling with a bug and trying not to commit code that was broken.

## Reference

GitHub Repository: https://github.com/bradleykenny/comp335-project

*It should be noted that the repository has been set to private and permission to access will need to be requested from* bradley.kenny@students.mq.edu.au.

[1] "Socket Programming in Java", GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/socket-programming-in-java/. [Accessed: 29- Mar- 2019].

[2] "Simple Mail Transfer Protocol (SMTP)", GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/simple-mail-transfer-protocol-smtp/. [Accessed: 30- Mar- 2019].

[3] "Socket (Java Platform SE 7)", Oracle Docs. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html. [Accessed: 30- Mar- 2019].