DOS DISTRIBUTED SYSTEMS JOB SCHEDULER

Dominic Christie (45189722), Ahmad Sohial Ahmadiar (45129223), Omar Ijaz Chaudhry (44148461)

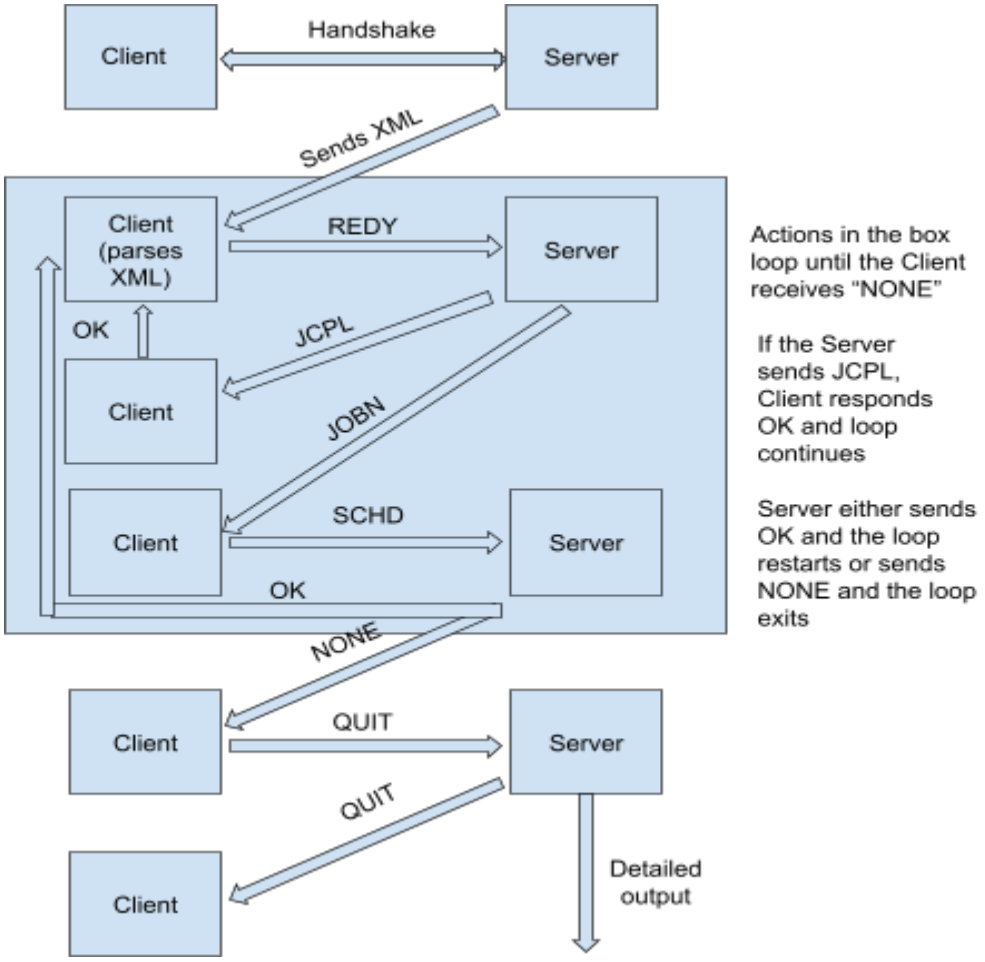
1. PROJECT GOALS

The overall goal of the project is to provide a cost-efficient allocator of server resources to complete server-provided jobs. Stage 2 will develop the work completed in Stage 1 by implementing multiple different algorithms to schedule jobs to the server that is most appropriate for the task.

The goal of Stage 1 is to build the bones of the client that will ensure that jobs can be passed from server to client and then subsequently scheduled in a way that jobs are passed to the largest server. This ensures that all jobs will be completed without resource allocation issues. This is unfortunately not maximally efficient, and that is what Stage 2 will develop upon; to ensure that the jobs are scheduled correctly to the most appropriate rather than largest server.

2. SYSTEM OVERVIEW

The client authenticates itself to the server, which in turn responds with details about the servers available. Once this pseudo-handshake is complete, the server then passes jobs to the client which schedules them in a more efficient manner, to be run on the server. Following the completion of the jobs on one or more servers, details are provided regarding the cost of the jobs and their success or failure.



3. DESIGN

3. 1. Design Philosophy

Program in a manner which is cognisant with the guidelines and is of sufficient clarity to be understood by third parties. Utilise modularisation and local/global variables to ensure the code is easily modifiable. Upload to a central repository (Github) to track progress.

Considerations	Constraints
How to translate the XML provided into usable variables that could be implemented in scheduling.	The minimal time to learn how to program distributed systems and to do the assignment.
How the code will interact with the relevant operating system, in this case: Linux (Ubuntu).	Difficulties in ascertaining specific code implementations based on resources provided.
How to ensure that our code is readable to markers and third-parties.	Limitations of Virtualbox's compatibility with different hardware/software.
The DOS Distributed job scheduler completes the task that was programmed to complete.	Dealing with libraries or structures which were deprecated such as DataInputStream.

3.2. COMPONENTS AND FUNCTIONALITY

Name of Component	Functionality
ClientTest()	Setting up the socket, for the input and output streams.
Communicator()	Authenticates the client to the server, calls the parse() function and then schedules jobs.
quitCommunicating()	Sends Quit to the client, if it gets the quit message in response, the input/output and server closes.
messageSend()	Send a message using DataOutputStream and then flushing it.
messageReceive()	Receiving a buffered message and returning a string.
largeServer()	Finding the largest server based on

	core count, and returning it.
parse()	Parsing the XML file in the correct form and translated it to local variables.
main()	Creates an instance of ClientTest() which then runs the Communicator().
Folder.Server	Server Class to initialise the server properties such as corecount, memory, disk, bootupTime, etc.

4. IMPLEMENTATION

4.1. Implementation of Libraries

Our program imports a number of libraries, essential to its successful running:

- java.io.*
    - Java IO is the library that allows programmers to utilise basic input and output. This is especially important for our program given our requirement to have constant communication with other programs i.e. ds-server.
- java.net.*
    - Java.net is responsible for allowing programmers to utilise structures such as Sockets, which are evidently crucial to this project. It is the way in which we initialize the server socket and ultimately communicate.
- java.util.*
    - Java.util is crucial for allowing the program to use ArrayLists, which let the programmer import the Server structure into an Array. Although only super-silk was used in this first Stage, the ArrayList will surely prove more useful in Stage 2, where multiple servers will be used.
- javax.xml.parsers.*
    - This is a self-explanatory library that allowed us to parse the XML, a crucial step in understanding the server properties.
- java.org.w3c.dom.*
    - This library allowed us to use a NodeList which assisted in the parsing of the XML and allowed us to hold variables in perpetuity within the List.

Our program initially relied on extensive code without structure, all contained within the main() function. Upon recollection of earlier Java units studied, modularisation and the use of functions, constructors, getters and setters was required. Modularisation dramatically improved the code's clarity. It also helped with the segregation of variables and thus the quarantining of values within each function. We

also moved the Server.java file to a separate location within the broader */src/precompiled* folder and then subsequently imported it.

4.2. IMPLEMENTATION OF EACH COMPONENT

4.2.1. Public class ClientTest()

This broad wrapping function contains the entire program. It also contains the global or universal variables that we did not want to initialise locally, such as the *Socket*, the *InputStream* and the *Output Stream*. After reading about issues online in the use of *DataInputStream* I chose to wrap it in a *BufferedReader*. I did not read about any such issues with *DataOutputStream,* I therefore initialised it without buffering. I also initialized the *msg* String which creates a universal String variable to be set to whatever message the Server sends. *Server ArrayList* and the integer variable *largeServer* were also created. Locally initialising these would have led to issues when attempting modularisation. I (Dom) developed and uploaded this to Github, although Omar and Sohial also came to these conclusions given they were discussed in the workshop.

4.2.2. Public ClientTest()

This function simply initialises the socket with the values of *localhost* (being 127.0.0.1) and the relevant port (being 50000). I then initialised the input and the output using a buffered *DataInputStream* and a non-buffered *DataOutputStream.* The function also utilises an exception catcher that prints any errors. I (Dom) developed and uploaded this to Github although my group members also worked this out.

4.2.3. Public void Communicator()

This crucial function makes essential use of the *messageSend* and *messageReceive* functions which are detailed further below. Conceptually, the function is rather simple. We pseudo-authenticate with the sending of *HELO*, and *AUTH Comp3100.* I use the term pseudo-authentication as there is no actual authentication, just a mere providing of a name. Upon conclusion of this exchange, the server will send across acknowledgement and the XML file, from whence the client replies with *REDY.*

The next chunk of code within the function schedules jobs. I set this up relatively easily however ran into issues relating to the treating of the JCPL message. I ultimately worked out that I can use an if/else conditional to address the JOBN first, if *msg* is not equal to JCPL. I also used a String Array to split up the message received from the server so that I could identify individual values such as the Job Count or whether it was JCPL/JOBN. I also implemented a while loop that addresses whether the Server has finished sending jobs; when it has, the quitCommunicating() function runs. I (Dom) developed and uploaded this.

4.2.4. Public void quitCommunicating()

Quit is just a very simple function implemented to stop communication when the method is run. This function was created after a discussion with the tutor after class.

4.2.5. Public String messageSend() AND Public String messageReceive()

These functions are relatively simple and essentially do the opposite role so I have chosen to address them simultaneously. The *messageSend* function simply writes a set of bytes to the *DataOutputStream* and then flushes the output stream. The *messageReceive* function simply receives the buffered input stream and then sets it to the universal String *msg*. Both functions are wrapped in exception handlers; my comments in the code note that I kept getting error messages without them. We all developed this together with the assistance of the tutor.

4.2.6. Public int largeServer()

This is an important yet relatively simple function. It essentially runs a loop and checks the coreCount of the servers based on the parsed XML. If a server has more cores than another it becomes the 'largeServer' which is ultimately returned by the function. I (Dom) developed and uploaded this with help in the workshops from the tutor.

4.2.7. Public void parse()

A conceptually basic function, however as my GitHub commits will show, I struggled to implement this. I ultimately received assistance from the tutor who fit the missing pieces of the puzzle. The function essentially grabs the data from the XML file and stores it in local variables in the function i.e. int b for bootup time. These variables are then accessible to the rest of the program when it calls parse() in the Communicator function. I (Dom) developed and uploaded this with considerable help from the workshops.

4.2.8. Public static void main(String[] args)

This core java function merely creates a new instance of clientTest and then runs the communicator function. This was developed by all of us independently.

5. REFERENCES

https://github.com/DomChristie15/client-side