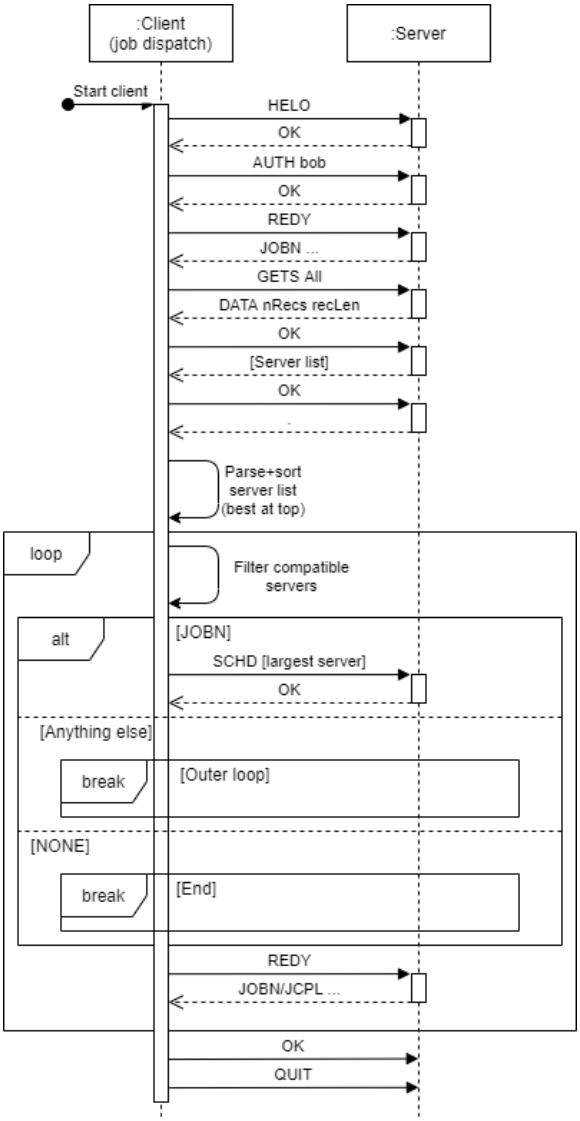
Client-Server Job Dispatch

Cory Macdonald (45445672), Georgia Gerrie (45423245), Matthew Aliwarga(45422117)

Introduction

Stage 1 involves designing and implementing a job dispatcher client to communicate with ds-sim servers and to appropriately dispatch jobs to the largest and least-busy server via an algorithm that determines as such. In general, the project will involve implementing a more sophisticated algorithm to dispatch jobs in a First-Fit (FF), Best-Fit (BF) or Worst-Fit (WF) fashion.



The goal of the project is establish communication with servers, find the largest server free and assign job tasks to this server. The goal in general is to implement an effective job scheduler for distributed systems.

System Overview

The system consists of a client-side and a server-side in which multiple servers can be found. The client establishes a connection with the server and requests for a list of servers, of which it then filters by what can handle the current job looking to be scheduled. The largest server is then selected for the current job. The server-side simulator receives the instruction to schedule the job on said server. It processes the job on that server and eventually returns with JCPL (job complete), which is ignored. This process is repeated until there are no jobs remaining and the server sends NONE, at which time the client exits.

Design

Design Philosophy

The design philosophy behind using Java SE 7's AutoCloseable interface was to ensure safety in managing all resources, such as the streams and the socket, such that they were automatically released when no longer required or in the case of an error. This helps reduce errors that may arise due to bad programming, and also makes the code easier to read.

Code readability was also a very important design focus, enforced by effectively using refactoring and appropriate naming. For example, the select server function, which conveys with relative ease that it will select a server to use based on the parameters provided, while the function itself uses names that are reasonably understandable within the context of this program. Readability was also improved by using comments where necessary to summarise the function of a particular block of code.

Considerations and Constraints

The primary constraint on this client was time. Provided more time, the client could have been made more versatile and the code probably could have been more clean. As it is, the design focused on completing the requirements for this particular stage of the assignment, for example by only implementing proper handling for NONE, JOBN and JCPL job types from the servers as any others aren't necessary in the configs being used.

However, the second stage of the assignment was also an important consideration, and as such, care was taken to avoid implementing anything that would cause significant issues down the line. As above, while only some job types are currently properly handled, the job class and code for receiving jobs can easily be extended so as to support any additional types that may be necessary for the next assignment stage. Another example of this consideration is that the select server function is very easy to replace with other algorithms, having been refactored out of the main code block and effectively isolated from the rest of the code.

Functionality of Simulator Components

send(DataOutputStream dout, String toSend), receive(byte[] data, int expectedSize, DataInputStream din), receive(byte[] data, String expectedString, DataInputStream din):
These were used to send and receive data from the server.

printErrorMessage(String expected, String actual), printErrorMessage(int expected, int actual):
Used to debug errors by printing out expected and actual values from within code where errors were expected to happen.

selectServer(Job current, ArrayList<Server> servers, DataInputStream din, DataOutputStream dout):
Used to select a compatible server to run a job on given a list of servers.

public static class Server:
This class was used to parse a server state (which was in String format) and give it an object representation in the code.

public static class Job:
Similar to the Server class, this was used to parse a string which contained Job information (usually a JOBN command, but could be otherwise).

public static class Data, parseCommand(String str):
This was used to store data from the DATA command passed from the server. parseCommand() would parse a string and return an instance of the Data class, which held the numbers (e.g. 450 and 124 from “DATA 450 124”).

Implementation

Cory (Send + Receive functions):

The send and receive functions were designed to simplify the process of just getting messages between the client and the server.

In the case of send, the function handles printing to the command line, writing to the output stream, and flushing the output stream so that process only takes one line. For receive, both implementations print the input and convert it to a trimmed String (removing any null bytes in case the exact size of the input couldn’t be known beforehand), which is then returned to be parsed appropriately if need be. The expectedSize implementation can be used most of the time, when only an inputs’ maximum length is known, otherwise expectedString can be used to confirm the server has sent exactly the message expected, like an “OK” after “HELO” (otherwise an error will be thrown).

Cory (Tuple):

The Java standard library has no in - built support for Tuples - as such, a simple, customized tuple class had to be designed for the project. The tuple contains two elements (and can otherwise be known as a Pair) - a String and an Integer, designed to contain a server type and a server ID respectively. This would be returned from the selectServer() function.

Cory (Server selection):

Server selection for any job is handled firstly by filtering the complete (already sorted) list of servers down to the list of servers that the job is able to be run on (based on the number of

cores the job requires, and that the server has). The first server is selected for the job. The job is then assigned with SCHD, the server sends OK, and the client indicates REDY again, receiving the next job and looping the whole process whilever the received job is JOBN (JCPL is ignored and simply responded to with REDY).

Matthew (Restructuring code to use the AutoCloseable interface from Java SE 7):

The implementation utilizes a Java SE7 feature known as the AutoCloseable interface (AutoCloseable (Java Platform SE 7) (oracle.com)). This interface allows the programmer to write code in the following format:

```
try (resource) {
    // ...
} catch (Exception e) {
    // ...
}
```

Any streams, such as those provided by the DataInputStream and the DataOutputStream, were initialized in the “resource” section of the code above, which allowed for multiple variables to be initialized within the parenthesis. The resources would be accessible from within the curly braces following it. Any resources initialized in the “resource” section must implement the AutoCloseable interface, which causes the implementing classes to have their close() method automatically called once the code running left the scope of the try block, or if an exception was called whilst code was running in the try block. This was implemented as a safer alternative as opposed to looking at each code branch and closing streams manually if an error had occurred, which would become more complex should an exception be thrown.

Matthew (Server class, Job class, parseCommand(), Data class):

These classes were created to create an object representation of their respective data, using data that was passed in as a String into its constructor. For instance, passing “DATA 450 124” to the parseCommand() function would lead it to construct a new Data object and return it by calling the Data constructor and passing 450 and 124 into it (after splitting the string based on spaces and parsing the numbers individually). Writing the code this way helped extract functionality for code reuse, which was seen multiple times throughout the rest of the code (for instance, when ArrayList of Servers were created to store Server information).

Matthew, Georgia (Error handling):

The entire program runs within a try / catch block with the pattern from above (see “Restructuring code to use the AutoCloseable interface from Java SE 7), allowing for easy error handling. If an error occurred in the code, an exception would be thrown with a custom message. This exception would be caught by the try / catch block in the main function, causing all resources initialized with the try / catch block to have their close() function called and for the program to print the error before closing.

Georgia (Error handling and Debugging):

Debug error printing functions were also created - both overloads of printErrorMessage() were created, one to handle Strings and one to handle integers.

References

https://github.com/Shadoweagle7/COMP3100-Project