COMP3100 Stage 1

Louie Coghill

April 4, 2022

46399089 - Session 1, 2022

Contents

1 Introduction

Stage one of the COMP3100 assignment prescribed the creation of a job scheduling client for a provided distributed system simulator (ds-sim). The client is to receive jobs from the ds-sim server, and schedule them for processing on simulated servers. This requires querying the ds-sim server for its available servers, and scheduling in some algorithmic way. Stage one requires that a Largest Round Robin job dispatching algorithm is implemented based on the servers CPU core count. The client is to be written in Java, and communicate with the ds-sim server using a TCP socket connection following the ds-sims communication protocol described in the ds-sim user guide [1].

2 System overview

The ds-sim server simulates a system of servers with their individual resources and costs specified by a configuration file. This file also specifies the number of jobs and their resource requirements that are to be scheduled by the client. Once all jobs are scheduled and have completed processing, the ds-sim server will announce to the client that there is no more work to be done with a "NONE" message, and both the client and server can terminate. As previously mentioned, all communication occurs over a TCP socket and transmission of byte encoding of Strings. Once a connection is made, the ds-sim protocol requires its own handshake before any scheduling can begin.

Once the handshake is completed, the client begins querying the server for jobs (and may receive job completion updates later on). The client must then figure out what servers are available for processing the job on the ds-sim, selecting an appropriate server to schedule the job to. A server is appropriate if it meets or exceeds the resource requirements of the job currently available, or will be available in the near future if they are currently reserved by another job. In the latter case, jobs join a queue waiting to be processed when the resources become available. Once the "NONE" message is sent by the server/received by the client, the client initiates the termination sequence for itself and the server.

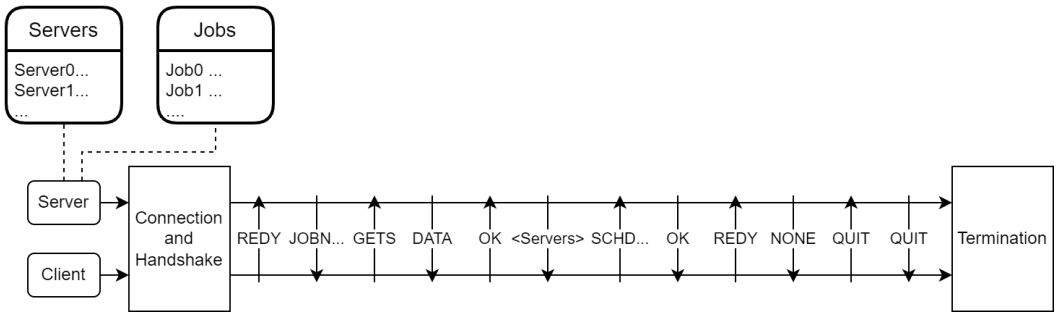

Figure 1: Basic server and client flowchart

3 Design

3.1 Largest Round Robin

Largest Round Robin scheduling specifies the following:

- Jobs are scheduled to all servers of the largest type
- The type is deemed largest by its CPU core count
- If there are multiple types of largest core count, the first type is to be used.

For example, if the available servers received by the client is the following

Server Type	id	Core count
Medium	0	4
Medium	1	4
Large	0	8
Large	1	8
AlsoLarge	0	8
AlsoLarge	1	8

Table 1: Example available servers

Jobs are to be scheduled exclusively to the "Large" type servers, as they have the highest core count (8), and occur before "AlsoLarge" which has an equal number of CPU cores and is therefore ignored. This selection of server type is to only occur once, meaning all jobs will alternate between being scheduled to "Large 0" and "Large 1" even if all of their cores become reserved for ongoing jobs.

3.2 Considerations

When receiving more complex messages from the server, that is, messages that contain information about more than one thing (e.g. server details), we are given a choice as to how it is stored and used. We can either work directly with the String which is very simple and straightforward, however harder to interpret when reading the code or create classes for these messages and store the data in instances. The latter option creates some overhead, however allows for much more readable and maintainable code with the addition of methods.

One more consideration we may make is how the client reads its incoming messages. As we will be working with a stream of data, it is almost impossible to determine how many bytes are being read in beforehand. This means the use of a byte buffer will be impractical as the buffer size must be determined prior to reading. Instead the ds-sim protocol offers the option to delimit messages by lines using the newline character creating many smaller messages.

3.3 Constraints

Unlike receiving a list of servers, jobs are sent/received one by one. Therefore the number of scheduling decision cannot be pre-determined and must continue until there are no more jobs to be scheduled.

4 Implementation

My client makes use a various Java packages for establishing a connection, receiving and sending messages, and for storing data where necessary.

4.1 Key java library imports

- java.net.Socket [2]

  The Socket package allows for application to application communication on both the same machine or on different machines and is used in my client to communicate with the ds-sim server. In addition to Socket, java.net.UnknwonHostException was imported as Socket can throw this exception.

- java.io.InputStreamReader [3] and java.io.BufferedReader [4]

  The InputStreamReader takes in the input stream from a Socket and allows for the Java program to read from the stream. The BufferedReader then takes the InputStreamReader and allows for readingt of individual messages that can be separated by a newline character (”
  n or byte value 10).

  The BufferedReader allows for reading incoming messages via a Socket. It is capable of parsing lines of messages delimited by a newline character (” n” or byte value 10). This allows for being able to build an undetermined quantity list of Strings without worrying about how many lines of messages are being received.

- java.io.DataOutputStream [5]

  DataOutputStream allows for the sending of messages via a sockets output stream.

- java.util.ArrayList [6]

  ArrayList is a package that allows for storing an undetermined number of objects or primitive variables in an array like structure and includes a number a methods and features such as automatic resizing of the underlying array.

4.2 Data structures

I have created two classes to represent certain data received by the client with the goal of making it easier to work with and allow for easier implementation of features down the line.

The first class created is the ServerResource class. The class constructor takes a line from the available server string (response to a GETS) and parses the line, recording the necessary information to the instance being created. This allows us to use get/getter methods rather than having to specify specific characters or indexes after splitting the line by white space making much more

readable code. Currently it only records the server type, id, core count, however can easily be changed to record new information should it be needed in the future.

The second class created is the Job class. Its constructor takes a String describing a job, parsing it and storing all the details in the instances own variables. Currently this class is not as useful or necessary as the previous ServerResource class, but does benefit the program with some methods removing the need to work directly with Strings when scheduling jobs.

4.3 Client structure

Once my client makes a connection with the server and performs the initial handshake, it creates two new lists to store jobs and servers in. It then sends "REDY" and enters a loop where at the end of each run another "REDY" is sent.

If the response to "REDY" begins with the substring "JOBN", adds the job to the list of jobs and checks whether the list of servers is empty. If it is, then it sends a "GETS All" message to the server to populate this list. Once the list is populated with all servers, it is then filtered down to a single type that has the highest CPU core count. By checking if the list of servers is empty, we can ensure that this operation is only performed once.

Now with a filtered list of servers, we now know how many servers we can schedule to, their type and ID for scheduling, and additionally the number of jobs we've received with the list of jobs. In each loop, we then schedule to the (number of jobs % number of servers) index of the server list giving us the round robin behaviour of scheduling to server 0, server 1, ... server n, and back to server 0.

After some number of loops, there will be no more jobs to schedule and a "NONE" will be received by the client. This breaks the loop in the client for job scheduling and sends "QUIT" to indicating that we are disconnecting to the server. Afterwards, we can close the socket before the client actually terminates.

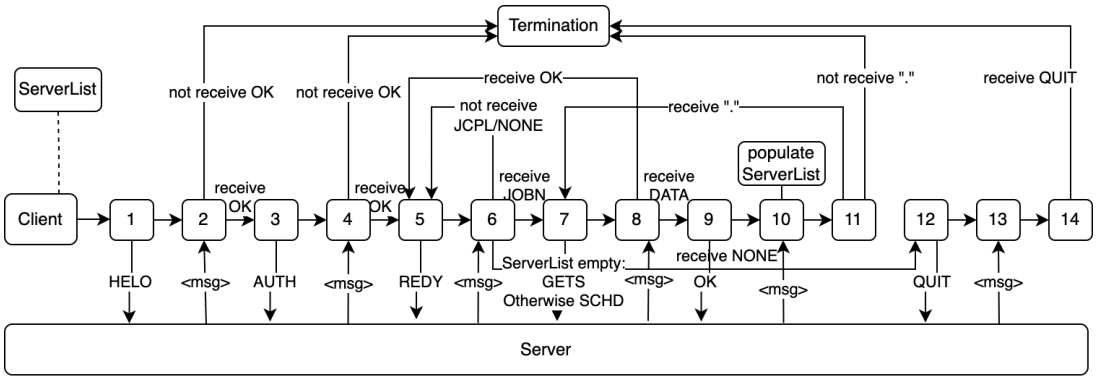

Figure 2: Client focused flowchart

My client Github link
ds-sim Github link

References

[1] "ds-sim: an open-source and language-independent distributed systems simulator User Guide" https://github.com/distsys-MQ/ds-sim/blob/master/docs/ds-sim_user-guide.pdf

[2] "Socket (Java Platform SE 8)" https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html

[3] "InputStreamReader (Java Platform SE 8)" https://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html

[4] "BufferedReader (Java Platform SE 8)" https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html

[5] "DataOutputStream (Java Platform SE 8)" https://docs.oracle.com/javase/8/docs/api/java/io/DataOutputStream.html

[6] "ArrayList (Java Platform SE 8) https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html