“All To Largest” Job scheduler

Group Members: Aydin Sumer (45437009), Jonathan Bui (45965188), Jonathan Skirtun (45982333).

Introduction:

This project is a client-side job schedular with the goal to connect to a job server and schedule all jobs to the first one of the largest server type, which is the one with the highest core count in the server list.

System overview:

For a full system workflow see appendix A figure 1.

The system uses byte streams to send and receive messages between client and server. To communicate with the server a handshake protocol is used to send and receive specific messages between the client and the server.

The program will use the GETS command to have the server send a list of server data. It will then parse this data into an array of server objects and sort in ascending order first by core count then name. It will then loop backwards from the end of the server to find the first of the largest core count server.

In the scheduling phase the client receives jobs from the server and will parse the response it receives. The client will schedule any job sent using the SCHD command. After all jobs are complete the server will send NONE response. The program will then tell the server to quit and close the socket and quit itself as well.

Design:

Communication with the server is achieved with the creation of a new socket object with a provided address and port number that the server runs on. An output stream to send messages and an input stream to read messages. String messages need to be converted to bytes to be sent to the server. For reading messages buffered reader is used to easily convert bytes to text. The handshake protocol is a specific sequence of predefined messages and responses.

To find the server with the largest core count we would need to get a list of the server data, store the data and sort. To get the data it was a choice between reading from the file or the GETS command. The gets command is easier to get the data sent from the server to the client than trying to read an external file. The data is then stored and sort by using a class object to store the data so we could use the java compare function along with sort utility to easily sort the data in ascending order. Then finding the first largest is done by looping backwards through the data array.

The SCHD command requires the job ID and server information to schedule a job, only the job ID is needs to be obtained during the scheduling component. To obtain the ID of a job the client needs to parse the job data received and look at the 3rd element of the message. After jobs are scheduled, the client will request and schedule more jobs by using a while loop and switch statement. A while loop was chosen as the number of jobs not known unless an external file is read. The switch statement was chosen over a nested *if* statement as it would be cleaner and only needs to test a single variable to determine the client response. When the server has no more jobs to schedule the loop breaks and the simulation closes.

Implementation:

Aydin Sumer oversees getting, storing, and sorting of server data to find the largest server. The goal of this component is to read the list of servers sent from the Gets command from the server. Then store the server’s type and core count in a class. Hold each server class in an array and sort using the arrays.sort utility in java. To do this a compare function is implemented in the server class and overridden such that it can return the compared value of the core count and the compared value of the server type if core counts are the same. So that the list of servers is sorted in ascending order of both name and core count. Then the program loops backwards from the end of the array to find the first of the largest as the list is in ascending order.

Jonathan Skirtun worked on the scheduling of jobs and closing the simulation in the ds-sim client. The client requests jobs from the server and parses the response so it can figure out the job type and the job number which it stores into an array of strings that is overridden each time it wishes to schedule a job. Its first job is retrieved from earlier in the program to start the scheduling process as it contains the necessary details, and a switch statement is used to determine the command to be sent to the server in response. The scheduling process loops sending the REDY command when it is waiting on a job or has scheduled a job already. The loop breaks upon receiving the NONE command signifying that the server has no more jobs to schedule. Upon exiting the loop, the client sends the QUIT command and closes the simulation while the server creates logs and error reports before shutting down.

Jonathan Bui led development on the initial connection and handshake. The goal of this component was to establish a connection with the server. The initial connection was established using the java socket class. The socket object was constructed with the server address and port number. Input and output streams were then created to send and receive messages between the client and server. The handshake starts with the “HELO” command, an “OK” command is received from the server as an acknowledgment. The client will then authenticate using the “AUTH command with the username as an argument, “OK” is again received. Finally, the “REDY” command is sent to the server to indicate the client is ready to receive jobs.

As communication between the client and server involves all three components of the client, the implementation involved a collaborative effort from all. Sending data used the DataOutputStream java.io class. String messages were converted to bytes using the String.getByte() java method. The java.util method write() was used to send the message, each method ending with \n as a sentinel. The java.util method flush() clears the stream to send the next message. Receiving messages involved wrapping a DataInputStream object with a BufferedReader. This allowed us to use the readLine() java.util method allowing for easy reading of text and handling of messages with multiple lines.

Appendix A:

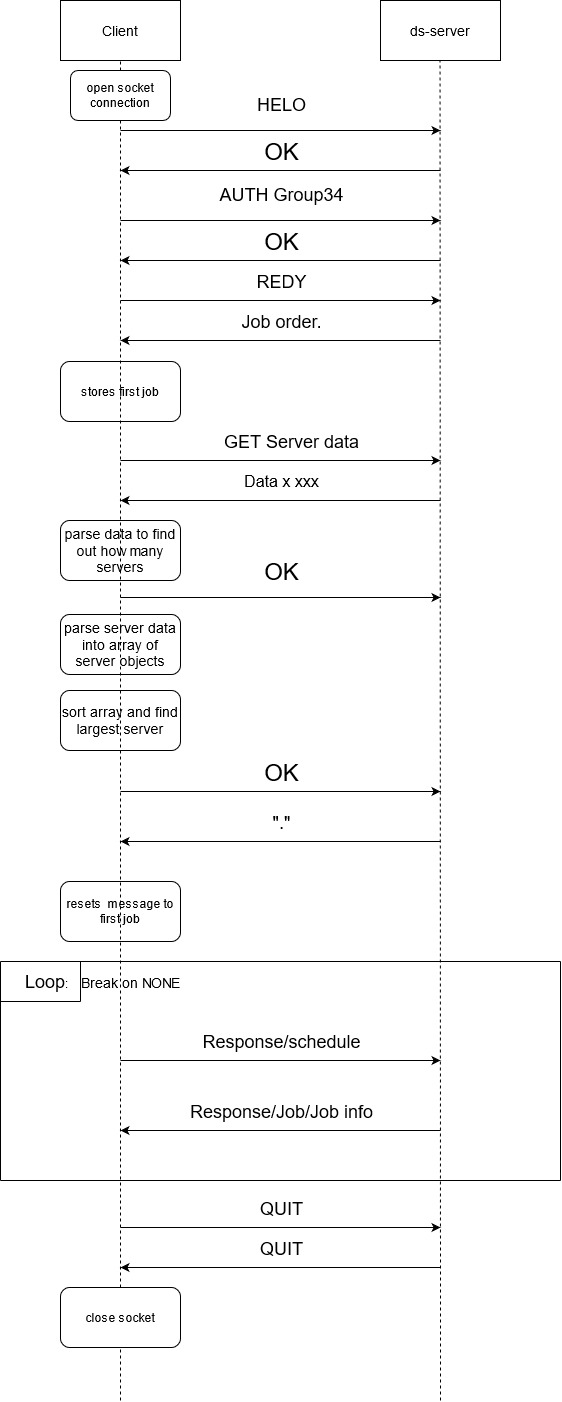


Figure 1- flow diagram

References:

https://github.com/a758/Group34ATL