“All To Largest” Job scheduler

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

Introduction (1/2 pages): What this project (focusing on Stage 1) is about, including the goal of the project and Stage 1.

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 (1/2 pages): high-level description of the system (both client-side simulator and server-side simulator with the focus being your client-side simulator), preferably, with a figure (your own, not one in ds-sim User Guide) showing the workflow/working of the system.

\*insert diagram of system. Here or references? \*

Sending and receiving messages with the server:

The system uses byte streams to send and receive messages between client and server. Specifically, the client uses a BufferedReader to read messages and a DataOutputStream to send messages.

Handshake protocol:

The handshake protocol involves the sending and receiving of specific messages between the client and the server. The client initiates the handshake with “HELO” and waits an “OK” response from the server. The client then authenticates with “AUTH (login)” and waits an “OK” response. The client will then send “REDY” and wait for jobs to schedule.

Gets command, getting the server list, largest server algorithm:

The system 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.

Scheduling and received jobs:

In the scheduling phase the client receives jobs from the server using the REDY command and will parse the response it receives into a string array determining its response though the use of a switch statement. The client will schedule any job sent using the SCHD command, job ID, server type and the server ID. Except for the NONE command which breaks the loop the client will send the REDY command to receive a new job to schedule. After the loop breaks the QUIT command is sent and the simulation ends with the server outputting housekeeping files.

Design (1 page): design philosophy, considerations and constraints, functionalities of each simulator component focusing on the client-side simulator.

Communication with the server is achieved with an output stream to send messages and an input stream to read messages. String messages need to be converted to bytes in order to be sent to the server. For reading messages the input stream was wrapped with a buffered reader to easily convert messages to text.

The handshake protocol starts with the creation of a new socket object with a provided address and port number. The client would then send the command HELO and wait for an OK response, the client will then send the AUTH command and wait for an OK response, finally the client sends the REDY command, and the handshake is completed. The handshake used conditionals to check received messages if they matched with the expected response.

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 find the first of the of the largest servers. To get the data it was a choice between reading the xml file that the ds-server produces or sending the GETS command to the server. It was decided to use the gets command as it is easier to get the data sent from the server then trying to read an external file. Then the data need to be stored and sort. For this it was easiest to use 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. Now to find the first of the largest is as simple as looping backwards through the data array.

The SCHD command requires the job ID and server information to schedule a job, as the server information is obtained during the GETS component only the job ID needs to be obtained during the scheduling component. To obtain the ID of a particular job the client needs to send the REDY command to the server and request a job, this data then needs to be parsed and stored into an array to obtain the job number. 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 over any other type of loop as the client would be awaiting input from the server rather than incrementing over an array or a set number of times. 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 (2 pages): brief description of any implementation specific information including technologies, techniques, software libraries and data structures used. How each of components/functions of your simulator is implemented including who is in charge of which function(s) and how they have led the design and development.

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.

References:

https://github.com/a758/Group34ATL