CMSC 417 Computer Networks **Fall 2015**

Programming Assignment 3

Assigned: September 29 *Due: October 11, 11:59:59 PM.* *Weight: 1.5x*

1 Introduction

In this project, you are tasked with creating a high-performance scheduler that will handle job requests from many clients and communicate with a compute server for each job. The scheduler will maintain a pool of TCP connections to the compute server, over which it will dispatch jobs for the clients. Additionally, each client has an associated priority value, which will be used by the scheduler to appropriately schedule jobs for all connected clients.

This setup is similar to how a lot of services are implemented on the Internet today. A front-end scheduler handles client requests and communicates with various back-end servers that work together to complete the request.

2 Protocol

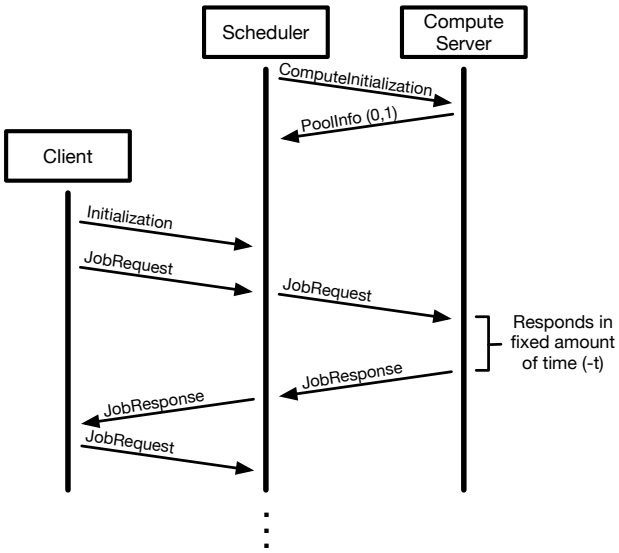

Figure 1: Example message sequence diagram for the assignment protocol. There is only one client, and the scheduler only has one connection to the compute server in its pool.

An overview of the protocol is shown in Figure 1. The protocol between the client, scheduler, and compute server will use five types of messages: ComputeInitialization, PoolInfo, Initialization, JobRequest, and JobResponse. The format of these messages will be explained in more detail later in this section.

The scheduler attempts to add a compute server connection to its pool by sending a ComputeInitialization message requesting to the compute server. If the scheduler is permitted to add

another connection the compute server will hold open the connection. Otherwise, the connection will be closed. In either case, a PoolInfo message will be sent and one compute unit will be "spent". The PoolInfo message specifies the number of connections the scheduler has open to the compute server before the most recent request and the number allowed. In this case, the PoolInfo message says that the scheduler has 0 open connections (which excludes this connection) and it is allowed to make a total of 1 connections.

After the client connects to the scheduler, the client sends an Initialization message that specifies its priority value for scheduling purposes. Next, the client sends a JobRequest to the scheduler. The scheduler in turn sends this JobRequest to the compute server on one of its pooled connections. The compute server will respond with a JobResponse after a fixed amount of time, which the scheduler will then forward back to the client. Afterward receiving the JobResponse, the client will send another JobRequest (and thus repeat this cycle). Note that, in general, the client may queue up multiple JobRequests to the scheduler instead of waiting for a JobResponse before it sends the next JobRequest.

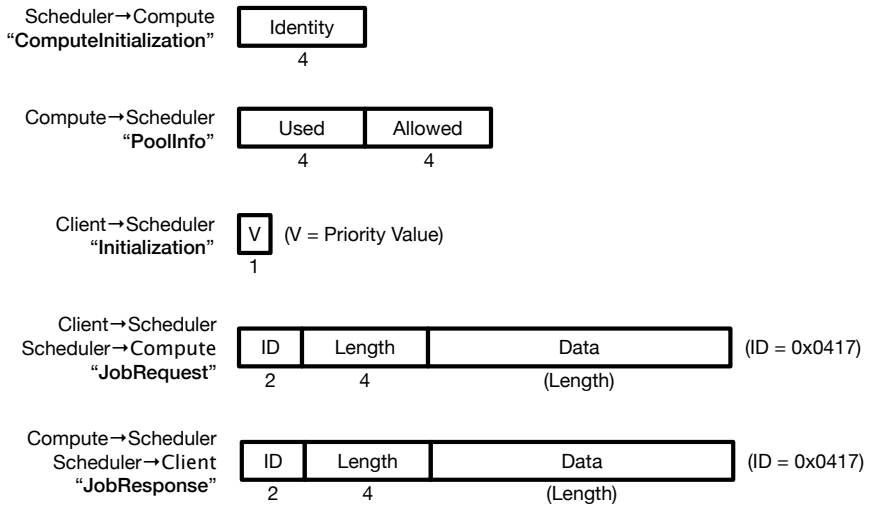

Figure 2: Format for messages used in the assignment protocol.

The formats for the three types of messages are shown in Figure 2. Additional details for each type of message are listed below.

ComputeInitialization

1. Identity: A 4-byte integer value in network byte order that corresponds to your Student ID.

PoolInfo

1. Used: A 4-byte integer value in network byte order that corresponds to the number of connections used, excluding the connection that the PoolInfo message was received on.

2. Allowed: A 4-byte integer value in network byte order that corresponds to the number of connections allowed for a given identity. No more connections will be accepted so long as Used is equal to Allowed.

Initialization

1. V: A 1-byte integer value that corresponds to the priority value of the client for use in scheduling jobs.

JobRequest and JobResponse

1. ID: A 2-byte integer value in network byte order that is set to the value 0x0417.
2. Length: A 4-byte integer value in network byte order that denotes the length of the Data payload in number of bytes.
3. Data: A payload that contains the data of either the request or response.

3 Client (Provided)

The client is a command line utility which takes the following arguments:

1. **-a <String>** = The IP address of the machine that the scheduler is running on. Represented as a ASCII string (e.g., 128.8.126.63). Must be specified.
2. **-p <Number>** = The port that the scheduler is bound listening on. Represented as a base-10 integer. Must be specified.
3. **-v <Number>** = The priority value of the client. Represented as a base-10 integer. Must be specified, and in the range of $[1, 10]$.

An example usage is as follows:

client -a 128.8.126.63 -p 41701 -v 5

The client will begin by sending an Initialization message to the scheduler, which contains the client's priority value (-v). Afterwards, the client will repeatedly: 1) send a JobRequest, and 2) wait for the JobResponse from the scheduler. Note that the client may have multiple outstanding JobRequests at any time at the scheduler.

4 Compute Server

The compute server is a command line utility which takes the following arguments:

1. **-p <Number>** = The port that the compute server will bind to and listen on. Represented as a base-10 integer. Must be specified.
2. **-n <Number>** = The number of connections supported by the compute server for a each identity. Represented as a base-10 integer. Must be specified, and in the range of $[1, 100]$.
3. **-t <Number>** = The time taken to finish processing each job. Represented as a base-10 integer with the units of milliseconds. Must be specified, and in the range of $[0, 5000]$.

An example usage is as follows:

compserver -p 41702 -n 25

The compute server listens for incoming TCP connections from schedulers, allowing each scheduler to connect to it at most (-n) times. After accepting an incoming connection, the compute server waits for a ComputeInitialization message from the scheduler and examines the identity value contained in the message. The compute server responds with a PoolInfo message, specifying the number of connections that identity has open (excluding the current connection), and the total number permitted for any identity to have open (-n). If the identity specified in the ComputeInitialization message already has (-n) connections open, then the compute server will close the connection. Otherwise, the compute server will maintain the connection.

The compute server waits for JobRequests on each connection. When it receives a JobRequest, the compute server will spend (-t) time processing the job and then reply back with a JobResponse. The compute server will operate on a single job on each connection at a time; if there are multiple JobRequests sent to the compute server on a single connection, it will handle them in a sequential fashion.

The compute server will randomly close individual connections from time to time.

You do not have to implement the compute server. We will provide an instance that you may connect to.

5 Scheduler (You!)

The scheduler will be a command line utility, which takes the following arguments:

1. **-p <Number>** = Port that the scheduler will bind to and listen on. Represented as a base-10 integer. Must be specified, with a value > 1024.
2. **–csa <String>** = The IP address of the machine that the compute server is running on. Represented as a ASCII string (e.g., 128.8.126.63). Must be specified.
3. **–csp <Number>** = Port that the compute server is bound listening on. Represented as a base-10 integer. Must be specified, with a value > 1024.

An example usage is as follows:

scheduler -p 41701 –csa 128.8.126.63 –csp 41702

The scheduler begins by establishing a pool of connections to the compute server. The scheduler will connect to the compute server and receive a PoolInfo response, which specifies the number of used and allowed connections the scheduler may open and add to the connection pool. The scheduler may build the pool at initialization or on demand, but must be aware that occasionally the compute server may disconnect scheduler. It is the responsibility of the scheduler to initiate new connections to the compute server as needed in order to maintain a full connection pool.

Afterwards, the scheduler starts to listen for incoming connections on a TCP socket bound to the port specified in the command line arguments. For each client that it accepts, it first expects to receive an Initialization message sent by the client which specifies the client's priority value. Afterwards, the scheduler waits for the client to send a JobRequest.

The goal of the scheduler is to handle JobRequests in a manner that is similar to weighted fair queueing (WFQ). Let us define the set of all clients currently connected to the scheduler as

C. Additionally, let us assume that this set is stable in that no new clients are connecting and no existing clients are leaving. Then, each client $C_i \in C$ (with priority value V_{C_i}) can expect a proportion of jobs sent to the compute server of:

$$\frac{V_{C_i}}{\sum_{C_j \in C} V_{C_j}}$$

For example, if there are two clients with $V_{C_1} = 2$ and $V_{C_2} = 1$, then C_1 should account for 67% of the JobRequests forwarded to the compute server by the scheduler.

The expected proportions of JobRequests to be met in the timeframe of roughly 10 times the job completion time for the compute server (specified by -t).

When the scheduler receives a JobResponse message, it must print out a single line in the format of:

[<Timestamp>] JobResponse <JobResponse Data>

<Timestamp> must be the time in milliseconds since the epoch. <JobResponse Data> must contain the entire payload of the JobResponse printed out in hexadecimal format and prefixed with a "0x".

6 Grading

Your project grade will depend on the parts of the project that you implement. Assuming each part has a "good" implementation, the grades are as follows:

Grade	Parts Completed
A	Scheduler is near optimal and correctly implements WFQ
B	Scheduler uses all compute capacity available, but does not correctly implement WFQ
C	Scheduler satisfies protocol, fails to use all compute capacity, but does implement WFQ
C-	Scheduler satisfies protocol, fails to use all compute capacity, and fails to implement WFQ

Extra Credit Rather than weighted fair queueing, an alternative strategy students may implement for extra credit would be to always service a higher priority clients before serving any lower priority clients. Unfortunately, this implies that a single high priority client can starve any other clients from being serviced. This alternative stategy will be used when **–aging 0** is passed as a command line argument.

For additional extra credit, students may implement aging on top of the aforementioned strategy. When the **–aging <seconds>** option is provided to the scheduler (with seconds > 0), requests that are queued at the scheduler should gain 1 priority level for every (–aging) seconds that pass. This will prevent the starvation of low priority clients, as their requests will eventually reach priority level 11 (which is higher than any client may specify).

7 Additional Requirements

1. Your code must be submitted as a series of commits that are pushed to the origin/master branch of your Git repository. We consider your latest commit prior to the due date/time to represent your submission.

2. The directory for your project must be called 'assignment3' and be located at the root of your Git repository.

3. You must provide a Makefile that is included along with the code that you commit. We will run 'make' inside the 'assignment3' directory, which must produce a 'scheduler' executable also located in the 'assignment3' directory.

4. You must submit code that compiles in the provided VM, otherwise your assignment will not be graded.

5. Your code must be -Wall clean on gcc/g++ in the provided VM, otherwise your assignment will not be graded. Do not ask the TA for help on (or post to the forum) code that is not -Wall clean, unless getting rid of the warning is the actual problem.

6. You are not allowed to work in teams or to copy code from any source.