CS3103 Programming Assignment 2

AY 2021/2022 Sem 1

Important deadline

- Due: **Nov 19, 2021, 23:59**
- Total Marks: 80 points (20% of Finals Marks)
- Late submission: 20% penalty per day
- Refer to this link for FAQ.
- **Note: This assignment can be done individually or in group of at most 2.**

Submission:

- Submit to folder “Programming-Assignment-2“.
- Your file should have the name “ Your tarball should be named "cs3103-assignment2-StudentNumber.zip” (if done individually) or “cs3103-assignment2-StudentNumber1-StudentNumber2.zip” (if done in group of 2).

Contact:

- TA : Chahwan Song (songch@comp.nus.edu.sg)
- Office hour : 3~4PM, 1st Nov (Mon). To be recorded.

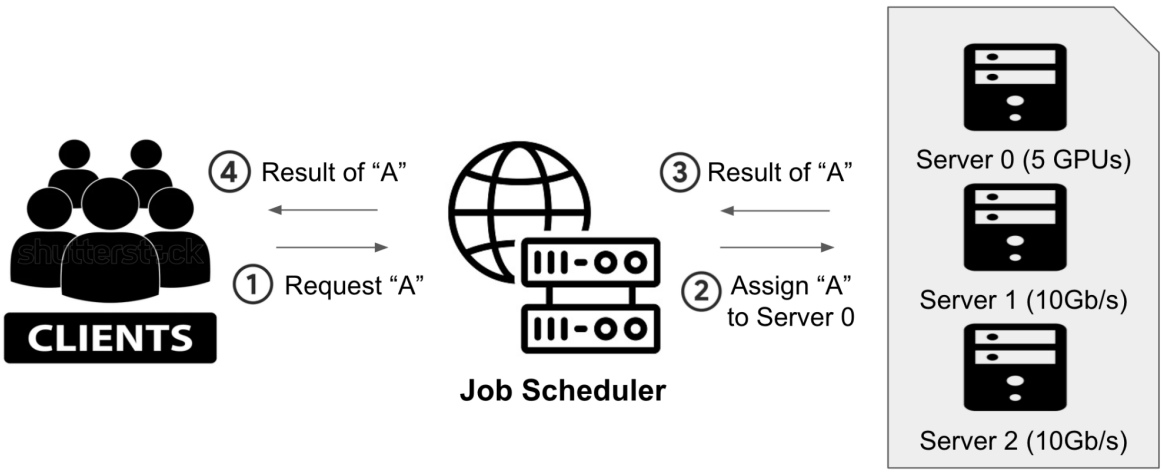
1. Introduction

In this assignment, you will implement a job scheduler which assigns the requested job to an appropriate server. The system receives job requests (e.g., computing tasks) and passes them to the computing cluster. The cluster consists of multiple servers with a fixed computing capacity. Different servers can have different computing capacities. The objective is to schedule the given job assignments to different servers so as to reduce overall job completion time.

2. System Description

2.1. Overview

In this section, we outline the logical flow on how your job scheduler interacts with clients and servers. The overview is depicted in the figure below.



The system consists of a set of clients, a cluster of servers, and your job scheduler. First, clients send the job requests to the scheduler (step 1). Then, the job scheduler makes a decision and assigns the request to a specific server (step 2). The server receives and processes the request, then returns back its result to the job scheduler (step 3). At the end, the client will receive the result (step 4).

In this assignment, ***you will design your own policy in the job scheduler***, which decides which server to forward the received job requests to.

2.2. Objective

The job completion time (JCT) is measured as the time the job request is received by the job scheduler (Step 1) to the time the job is completed (Step 3). In this assignment, the objective goal is to ***minimize the 50th (median) and 95th (tail) percentile of the JCTs.***

2.3. Logical Flow

The logical flow of the clients, job-scheduler, and servers are as follow:

Clients → Job Scheduler

Clients send requests to job scheduler, which include the information <“*filename*”, “*job-size*”>. Literally, filename is an identifier of job request, and job-size is the size of job. Obviously, a larger job-size takes longer time to process.

Note that job-size is *NOT always given* because the clients even do not know the size in some cases. Even if the job size is given, the job scheduler does not know the sizes in advance (i.e., scheduler is not an oracle). Rather, it knows only when the corresponding request arrives at a certain time. For instance, consider a batch of job requests will be sent with the sizes at t = 2. Then, the job scheduler does not recognize the requests until t = 2, but will know the requests with the sizes at t = 2, then will make a decision.

In this assignment, we consider three cases: (1) *none of the information is given (i.e., 0% information), (2) some of the information is given (i.e., 50% information), (3) full knowledge (i.e., 100% information).*

Job Scheduler → Servers

Once it receives a request, it decides where to forward the request. The forwarding information will include <"*servername*", "*filename*", "job-*size*">. Each server has its own identifier (servername) and capacity. If a server has computing capacity X and Y concurrent jobs, it will be able to process at X/Y unit of jobs-size per second. For example, if the server's processing capacity is 100, then it takes 10/100 sec = 0.1 sec to process a single job of size 10. Alternatively, a server with processing capacity 100, can complete the processing of 10 jobs with size of 10 units.

The job scheduler knows the list of servers, but their capacity is *unknown*. Moreover,a job request *DOES NOT* need to be sent immediately once it arrives at the job scheduler. A job can be scheduled and be queued to be sent later.

Servers → Job Scheduler

Once the job request is completed by a server, the result is sent back to the job scheduler. The information includes <*F*"*filename*"[1]>. From this information, the scheduler can calculate the job completion time and infer how many requests are still being concurrently processed at any server based on the scheduling decisions it has taken previously.

Assumptions

For simplicity, we make a few assumptions in this assignment:

1. No Name Duplication: Both "*servername*" and "*filename*" will not be duplicated.
2. No Distributed Processing: A job has to be processed completely on a single server.
3. No Preemptive Scheduling: Once a request is assigned to a server, it cannot be paused or replaced until it completes.

3. Simulator of Job Scheduling System

To simplify the assignment, we provide a simulator that implements the skeleton code for the logical flow. Our simulator consists of two programs: (1) a single client+server program that implements both the client & server, and (2) the job scheduler. The two programs communicate using a TCP socket.

The client+server program serves as clients and servers, generating requests, processing them, and sending results to the proxy. This program will be given as an `executable file compiled in python3 (>=3.6), available on Linux (Ubuntu 18.04 or later)`. Therefore, what you need to design and develop your own program is the job scheduler. The skeleton code for the job scheduler program will be given in `python3` and `C++`.

Note that we DO NOT use python2 in this assignment. Also, this simulator may not run on Windows. You can utilize xcne1.comp.nus.edu.sg as a verified testbed.

[1] The leading alphabet *"F"* is to identify the completion signal.

3.1. Files

First, we introduce the files to be given in the assignment.

Skeleton Job Scheduler Codes

We provide a template code of job scheduler, in `python3` (**`jobScheduler.py`**) and `C++` (**`jobScheduler.cpp`**). Both include the same simple job scheduling program. Note that one major difference is the input of the socket recv/send function: `C++` socket gets the char array, whereas `python3` socket gets only the binary string. Therefore, if you use `python3`, you need to encode/decode the string format (`utf-8`, by default) to send/recv data through socket.

Here is a list of helper functions:

- **`sendPrintAll(socket)`**
  From your job scheduler programs, for debugging purposes, you can ask the servers to print all job status by sending a message "`printAll\n`". This is implemented in the **`sendPrintAll()`** function in the template codes. Note that this is a kind of oracle-code. In evaluation, you cannot use the result from this.

- **`parseServernames(buffer, len)`**
  Before running a job scheduler, it informs you of the list of servernames.

- **`parseWithDelimiter(string, delimiter)`**
  This returns a vector of strings parsed by the given delimiter, e.g., **`'\n'`** or **`','`**.

- **`parser_filename(request), parser_jobsize(request)`**
  From the request string parsed by `parseWithDelimiter()`, we further parse the request message into "*filename*" and "*jobsize*".

- **`scheduleJobToServer(servername, request)`**
  This function assigns a given *servername* to the *request*. In order to send a request to a specific server, this is the method by which you must use.

Config Files

Our simulator provides two config files: (1) **`config_server`**, and (2) **`config_client`**. They are written in a plain text. Note that these config files allow you to make your own dataset and test environment and evaluate your job scheduling program. In the evaluation for grading, the config files will be removed so that your program cannot cheat :)

(1) **`config_server`** contains the information of servers. Each line (with delimiter **`'\n'`**) includes fields {*"servername", "capacity"*} with delimiter **`','`**. Here is the example config of three servers having capacity 100, 50, and 10:

```
S0,100
S1,50
S2,10
```

(2) **`config_client`** contains the information of requests. Each line (with delimiter '\n') includes fields {*"timestamp", "filename", "job-size"*} with delimiter **`','`**. Here is the example config of five requests having filesize all 10 and a sending time at 0 or 1:

```
0,NET0,10
0,NET1,10
1,AI0,10
1,ML0,10
```

In the config files, you can comment the lines by adding **`'#'`** in front of a line, similar to `python3`.

`Pickles and plot.py`

Once the jobs are all processed, the server/client program generates the pickles named as **`client.pickle`** and **`server.pickle`**. The former includes the starting timestamp of the job (i.e., timestamp at step 1), and the latter includes the completion timestamp (i.e., timestamp at step 3). Using these results, `plot.py` computes the job completion time (JCT) and plot histogram using python `matplotlib` library.

3.2. Time Granularity

One may wonder how the simulator generates the timestamp (and measures JCT), and how much it guarantees the accuracy of the timestamp. Indeed, our client/server program updates all status in every 50ms by default. So, the timestamp also has a granularity of 50ms.

To guarantee the time accuracy, we restrict the number of servers and concurrent jobs per server to 10 and 200. Also, we restrict the number of requests per second to 200. Note that there is no restriction on your own job scheduling program. But, the time taken for decision making and forwarding requests will be accounted for in JCT. This time cost will be ignored if your scheduler decides less than 50ms, due to the given time granularity.

3.3. How To Run The Template Code

[Step 1] Setup servers & clients, and socket standby.

You can run **`server_client`** program to configure client's requests and servers and standby to run socket programming with a job scheduler. You may choose the port number and the ratio of jobs whose sizes are given.

For port number 12345 and 100% of jobs are sent with their sizes (i.e., full knowledge), you can use this command:

```
$ ./server_client -port 12345 -prob 100
```

On the other hand, you can make zero-knowledge job-sizes config by setting `-prob 0`. In this case, the job scheduler will receive the messages with `jobsize = -1`, e.g., "`{filename, -1}`". In the evaluation, we will use three probabilities: 0, 50, and 100.

[Step 2] Run a job-scheduling program.
You can run your own job scheduling program. Here, suppose we use the template codes. If you run in Python3, you can use the following command:

```
$ jobScheduler.py -port 12345
```

If you use jobScheduler.cpp, first you need to compile the cpp file. Its first argument is the port number. You can use the following commands:

```
$ g++ -Wall -std=c++11 jobScheduler.cpp -o jobScheduler.out
$ ./jobScheduler.out 12345
```

4. `To-Do` Tasks

There are functions you need to modify for this assignment (you can make additional changes to any part if you want or need to, as long as the program logic is not broken) are as follow:

1. **getCompletedFilename(filename)**

   Once a server completes a job, it returns the result message to the client. This function catches the result message, which is *"F{filename}"*. You should use the information on the completed job to update some statistics to drive your scheduling policy.

2. **assignServerToRequest(servernames, request)**

   This function assigns one of the servername to the requested job. Given the list of servers, which server do you want to assign to this request? You can use a global variable or add more variables to make your algorithm.

The given skeleton code implements a default decision of allocating all jobs to the same server.

5. Evaluation

Test Cases:

We will provide two set of test cases,

1. **Uniform** job size and computing capacities (2 files)
2. **Uniform** job size and **uneven** computing capabilities (2 files)

3. **Uneven** job size and computing capacities (2 files)

with changing the amount of given knowledge (e.g., **0%, 50%, 100%**) as explained in Step 1, Section 3.3. In the `assignment2.zip`, we provide 6 pairs of config_server and config_client files. Note that in the evaluation, your job scheduler cannot directly read the config files, but should make a decision based on the request messages from clients.

Grading:

[20pt] Code stability and algorithm description.

[30pt] Successful compilation and running for known test cases 1, 2, and 3.

[30pt] Unseen test cases.

In the evaluation, we will give the score based on the following dimensions:

1. JCT: **50th** percentile of JCT, and **95th** percentile of JCT
2. Knowledge of jobsize: **100%** (full knowledge), **50%**, **0%** (no knowledge)

Submission:

1. jobScheduler.py or jobScheduler.cpp
2. A README file (.pdf) that explains your algorithm