Assignment 4: A Simple Parallel Web Server

Due: Tues April 4, 11:59PM EST

100 points total

Overview

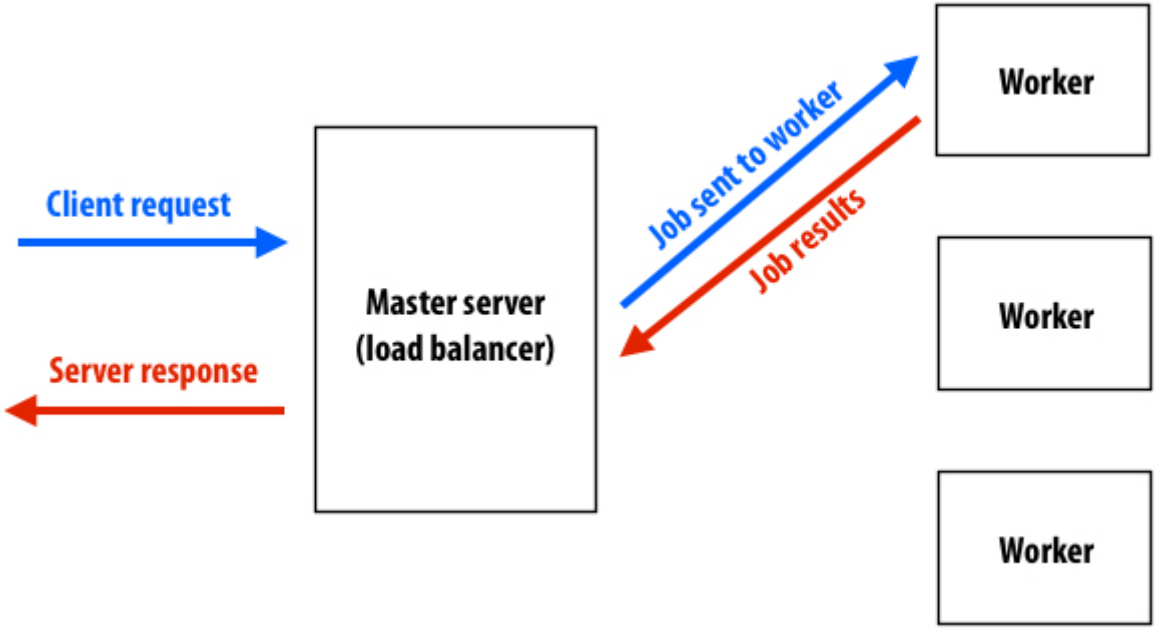
In this assignment you will implement a parallel server that uses a pool of machines to respond to a stream of input requests. Your goal is to respond to all requests as quickly as possible. That is, to minimize your server's *response time*. However, because running many servers can be costly, your server will have the capability to elastically adapt to variations in request stream load. A good implementation of your server will efficiently take advantage of all the processing resources in a single node, and also use more machines under times of high load (to minimize response time), and use fewer machines in times of low load (to minimize cost).

Basic Server Architecture

Your server will consist of a "master" node responsible for receiving all input requests, and a pool of "worker" nodes that perform the costly work of executing jobs triggered by client requests. Handling a request will behave as follows.

1. The server's master node will receive a request from a **client**.
2. The server's master node will select one or multiple worker nodes to carry out jobs related to the request, and send those jobs to these worker(s).
3. The worker node performs the jobs it is assigned by the master, then reports job results back to the master.
4. The master nodes collects results from worker(s), and then uses the result to respond to the original client request.

A simple diagram of this setup is shown below. In this assignment, the master node and each individual worker node in your server configuration will be hosted by different machines in the `latedays.andrew.cmu.edu` cluster.



The Master

The master node is responsible for interpreting incoming requests and generating jobs for workers. In this assignment, the master is a **single threaded process** that works in an "event-driven" manner. (You should not modify it to be multithreaded.) That is, the master process will call two important functions when key events happen in the system. Your job is to implement both of these functions:

```
void handle_client_request(Client_handle client, const Request_msg& req);
void handle_worker_response(Worker_handle worker, const Response_msg& resp);
```

`handle_client_request` is called by the master whenever a new client request arrives at the web server. A request is provided to your code as a dictionary (a list of key-value pairs: see `req.get_arg(key)`). Your implementation will need to inspect the request and make decisions about how to service the request using worker nodes in the worker pool. `client_handle` is a unique identifier for this request to your server. It is guaranteed to be a unique identifier for *all outstanding requests*.

The second function, `handle_worker_response` is called whenever a worker node reports results back to the master. The response of a worker consists of a tag and a string (see `resp.get_tag()` and `resp.get_response()`).

The responding worker node is identified by `worker_handle` (each worker node is given a unique handle which your master learns about in `handle_new_worker_online` -- see "Elasticity" section below.).

In order to implement your master, you are provided the following library functions.

```
void send_client_response(Client_handle client, const Response_msg& resp);
void send_request_to_worker(Worker_handle worker, const Request_msg& req);
```

As you might expect `send_client_response` sends the provided response to the specified client. This client handle should match the client handle provided in the initial call to `handle_client_request`. `send_request_to_worker` sends the job described by the key-value pairs in the request object to a worker. Assuming that you have implemented your worker node code properly, the master, after calling `send_request_to_worker` should expect to see a `handle_worker_response` event in the future.

The provided assignment starter code provides a bare-bones server implementation. It is commented in detail to help get you started.

The Worker

You are also responsible for implementing a worker node. To do so, you will implement two functions that are called by a worker process.

```
void worker_node_init(const Request_msg& params);
void worker_handle_request(const Request_msg& req);
```

`worker_node_init` is an initialization function that gives your worker implementation the opportunity to setup any required data structures (if you need any). `worker_handle_request` accepts as input a request object (e.g., a dictionary) describing a job. The worker must execute the job, and must send a response to the master. It will do so using the following two library functions.

```
void execute_work(const Request_msg& req, Response_msg& resp);
void worker_send_response(const Response_msg& resp);
```

`execute_work` is a black-box library function that interprets a request, executes the required work in the calling *thread of control* and populates a response. Your worker code is then responsible for sending this response back to

the master using `worker_send_response`.

At this point you may be wondering... since you gave me `execute_work`, what is there to do in the worker? In this assignment, your worker process will be executing on a machine with **two hyper-threaded, six-core CPUs**. Therefore, simply calling `execute_work` from within `worker_handle_request` will only use one execution context (of the possible 24) in the system! A good solutions to this assignment will need to effectively utilize all cores on all the worker nodes.

Elasticity

In addition to managing the assignment of request processing to worker nodes, your master node is also responsible for determining how many worker nodes should be used. To add elasticity to your web server, you will implement the following functions:

```
void master_node_init(int max_workers, int& tick_period);
void handle_tick();
void handle_new_worker_online(Worker_handle worker, int tag);
```

`master_node_init` allows to initialize your master implementation. The argument `max_workers` specificies the maximum number of worker nodes the master can use (requests beyond this limit will be denied). The system expects your code to provide a value for the argument `tick_period`, which is the time interval (in seconds) at which you'd like your function `handle_tick` to be called during server operation. `handle_new_worker_online` is called by the master process whenever a new worker node has booted as is ready to receive requests. (See below for details of how to request more worker nodes.)

Finally, your master implementation can request new worker nodes to be added to its pool (or request that worker nodes be removed from the pool) using:

```
void request_new_worker_node(const Request_msg& req);
void kill_worker_node(Worker_handle worker);
```

After calling `request_new_worker_node`, the system (at some point in the future) will notify your master that the new worker node is ready for requests by calling `handle_new_worker_online`. Booting a worker is not instantaneous. The requested worker node will not become available to your server for about a second. In contrast, `kill_worker_node` will immediately kill a worker. The worker should not be sent further messages after this call,

and any outstanding tasks assigned to the worker node are lost. **THEREFORE, IT WOULD BE UNWISE TO KILL A WORKER NODE THAT HAS PENDING WORK.**

The Incoming Request Stream

To drive your web server, we provide a request generator that plays back traces defined in the `/tests` subdirectory of the assignment starter code.

Your server must respond correctly to several types of requests. As stated above, your worker nodes will use calls to `execute_work` to process requests, treating its internals as a black blox. However, you'll need to understand the workload created by each type of request to make good scheduling decisions. We'll describe the nature of the different requests below, but it's likely you may want to look at the functions called by `execute_work` in `src/asst4harness/worker/work_engine.cpp`.

418wisdom: This request invokes the "418 oracle", a highly sophisticated, but secret, algorithm that accepts as input an integer and uses it to generate a response that conveys great wisdom about how to succeed in 418. Given the sophistication of this algorithm, it is highly *CPU intensive*. It requires very little input data as so its footprint and bandwidth requirements are negligible. The algorithm does approximately the same amount of work for every invocation, so running times will be argument independent.

projectidea: This request invokes another highly sophisticated oracle that, given an integer, provides suggestions for final projects. The algorithm used has a working set that is about 14MB, just fitting within the 15MB L3 cache of the processors in the `latedays` cluster. (Hint, I tell you this, because it might be important.) The algorithm does approximately the same amount of work for every invocation, so running times will be argument independent. **Also, since students really want to do great projects, the benefit of responding to projectidea requests quickly is signifcantly higher in the grading rubric than all other requests.**

tellmenow: This request is an automatic "office hours" service for Assignment 4. Students want help, and they want it now, so the grading rubic sets a very strict response latency requirement (150ms) for tellmenow requests. Luckily, tellmenow requests are very cheap, requiring only a few CPU ops to process.

countprimes: This request accepts an integer argument `n` and returns the number of prime numbers between 0 and `n`. The task has similar workload characteristics as 418wisdom. It is *CPU intensive* with little to no bandwidth or memory footprint requirements. However unlike 418wisdom, the runtime for countprimes requests is variable depending on the value of `n`. Smaller values of `n` result in cheaper requests.

compareprimes: This request accepts as input four arguments: `n1`, `n2`, `n3`, `n4`. It responds with the string "There are more primes in the first range" if there are more primes in the range `n1-n2`. Otherwise, it responds with "There are more primes in the second range". This request is unique in that your server will need to craft a response from the results of four calls to the library function `execute_work` (For more detail, see function `execute_compareprimes` in `src/myserver/worker.cpp`, which you are free to modify). Except for compareprimes, for all other requests your code will simply forward the response produced by `execute_work` back to the client.

Getting Started on the new Latedays Cluster

Assignment 4 will be run on the `latedays` cluster.

The cluster contains 18 machines (1 head node + 17 worker nodes). Each machine features:

- Two, six-core Xeon e5-2620 v3 processors (2.4 GHz, 15MB L3 cache, hyper-threading, AVX2 instruction support)
  - **http://ark.intel.com/products/83352/Intel-Xeon-Processor-E5-2620-v3-15M-Cache-2_40-GHz**
- 16 GB RAM (60 GB/sec of BW)

You can login to the `latedays` head node `latedays.andrew.cmu.edu` via your andrew login. You will develop and compile your code on the head node, and then run jobs on the cluster's worker nodes using a batch queue. At the moment, you have a home directory on `latedays` that **is not** your andrew home directory. (You have a 2GB quota.) However, your andrew home directory is mounted as `/AFS`.

At this time you are not able to work out of your `/AFS` directory since that directory is not mounted when your job runs on the worker nodes of the cluster. (It is only mounted and accessible on the head node.) Please develop and test your code in /home/USERNAME/...

To build the starter code on `latedays`:

```
git clone https://github.com/cmu15418/assignment4
cd assignment4
module load gcc-4.9.2
make -j 4
```

Note that you may find it useful to add `module load gcc-4.9.2` to your `.cshrc` or `.bashrc` so that you do not have to manually load the module into your environment on each login.

To submit a job to the job queue, use the default submission script we've provided you: `latedays.qsub`.

```
./run_latedays.sh 1 tests/hello418.txt
```

The first argument maximum number of worker nodes to use in your server, and the test to run. So in this example, we're configuring your server to run with at most one worker node. The result is a job that will take two nodes on latedays cluster, one for the master, and one for the worker. After a successful submission, the script will echo the name of your job. For example, if your job was given the number 337 by the job queue system, the job would have the name:

```
337.latedays.andrew.cmu.edu
```

Now that you've submitted a job, you can check the the status of the queue via one of the following commands:

```
showq
qstat
```

When your job is complete, log files from stdout and stderr will be placed in your working directory as:

```
latedays.qsub.o337
latedays.qsub.e337
```

The job you submitted drives the server using a request stream defined in `tests/hello418.txt`. (The server in the starter code supports only one outstanding request and uses one worker node.) The request stream defined by `tests/hello418.txt` contains one request to the 418 oracle. The test file, shown below, contains a list of requests, the expected server response, and the time at which the request is issued during the test (relative to the test start time).

```
{"time": 0,    "work": "cmd=418wisdom;x=1", "resp": "OMG, 418 is so gr8!"}
{"time": 500,  "work": "cmd=lastrequest",   "resp": "ack" }
```

Take a look at the output of the run in: `latedays.qsub.o337`. As you can see, the test script produces a convienient report about the correctness and latency of each response from your server. Other request streams for testing are provided in the `tests/` directory of the assignment starter code. Tests used for assignment grading will also produce a report giving your grade for the test:

```
spawning master process on compute-0-15.local (0 of 2)
compute-0-11 compute-0-15
command was: '/home/USERID/asst4/spawner.sh 1 tests/hello418.txt'
Waiting for server to initialize...
Server ready, beginning trace...
Request 0: req: "cmd=418wisdom;x=1", resp: "OMG, 418 is so gr8!", latency: 1108 ms
Request 1: req: "cmd=lastrequest", resp: "ack", latency: 0 ms

--- Results Summary ---

[0] Request: cmd=418wisdom;x=1, success: YES, latency: 1108

*** The results are correct! ***

Avg request latency:  1108.04 ms
Total test time:      1.11 sec
Workers booted:       1
Compute used:         1.45 sec

No grading harness for this test
```

Since your server is running on multiple machines, the testing harness logs all output of the master and each worker to a `logs.337.latedays.andrew.cmu.edu` directory. For example, the log from the worker node running on hostname `compute-0-11.local` is dumped to the following file in the `logs` directory:

```
worker.compute-0-11.local.kayvonf.log.INFO.20150314-183417.41087
```

For convenience, `logs.337.latedays.andrew.cmu.edu/master.INFO` and `logs.337.latedays.andrew.cmu.edu/worker.INFO` will soft link to the master's log, and a log of one of the workers. (replace 337 with your own job id)

The maximum wall clock time for any job is currently limited to six minutes. (See the comment in `latedays.qsub` about changing the wall-clock time of your job if you need to increase it for temporary debugging.) This means that we will cut your program off after 6 minutes.

More Help

1. The code you will edit in this assignment is in the directory `src/myserver` of the start code. You may change anything you wish in the starter code in this directory, but you **should not change any other code**. We recommend that you begin by skimming through the provided starter code in `src/myserver/master.cpp` and `src/myserver/worker.cpp`.
2. Next, we recommend you improve your server implementation so that it correctly handles all the request streams located in the `tests/` directory. The starter code does not provide correct results for these tests.
3. Once you have a correct server, then consider how to improve the server's performance. Ideas might include using more workers or parallelism within a worker node.
4. At this point it is time to begin understanding the behavior of the traces. Understanding your workloads will be very important in this assignment.

Running the assignment on other GHC machines

It is also possible to run the assignment for development and testing purposes on GHC machines using the staff-provided `run_ghc.sh` script. This script might be useful while simply trying to get up and running and establish basic correctness, since you don't have to wait to get through the `latedays` queue.

```
./run_ghc.sh 1 tests/hello418.txt
```

To avoid colliding with other students on the machine you are logged into, we recommend using a unique port for communication. You can get a unique port using:

```
python scripts/port-for-user.py
```

This gives you a unique port. Then tell the assignment harness to use the port via:

```
./run_ghc.sh 1 tests/hello418.txt [MY_PORT_NUMBER]
```

This commandline is equivalent to the latedays qsub script discussed above. It will configure your server to use at most one worker node. The worker node will be launched as a separate process on the same machine. (Instead of on

a different machine). We note that the performance grading scripts are carefully calibrated for performance expectations on `latedays`. You should disregard any performance grading results if you are not running on `latedays`.

Assignment Rules

Although we offer you significant flexibility to implement this assignment however you choose, there are a number of rules we would like you to abide by in order to maintain the intended spirit of the assignment:

1. If you wish to take advantage of multi-core parallelism in this assigment you are to use pthreads. (Recall your use of threads for the web proxy assignment in 15-213.)
2. **We ask that you do not spawn multiple threads of control in your master node implementation. We'd like you to stay with the single-threaded, event-based design of the system.** You are more than welcome, and in fact encouraged (see point one above), to spawn threads to enable parallel execution in your worker node implementations.
3. You may only call `execute_work()` from worker nodes (do not duplicate its functionality in your own code to circumvent the rule). That is, you should think of the master node as a load balancer that is primarily responsible for scheduling/distributing work associated with client requests and potentially combining results from various worker nodes. The "heavy lifting" to service requests should be performed by workers. Implementing caching logic within the master node (and/or the worker nodes) to memoize the results of worker computations is acceptable.
4. Yes, there are faster ways to compute primes, but that is not what we want to your learn in this assignment. (We want you to focus on scheduling.) Please use the implementation in `execute_work()`.
5. **Yes, you are allowed to specialize your implementation for various traces. However, we challenge you to try to design a scheduling policy that does not have any trace-dependent behavior. We will consider the possibility of extra credit for students that have particularly elegant, general solutions. Explain your solution clearly in your writeup if you want to be considered (also consider generating your own traces to evaluate it in ways we did not think of!).**

Grading

This assignment is worth 100 points. The assignment writeup is worth 20 points. The remaining 80 points depend on the correctness and performance of your server when run within our test harness on `latedays`. While the correctness of your server will be tested using traces not provided to you, the performance of your server will be graded using a **max worker nodes = 4 configuration of the server**, and using the following traces in the assignment starter code:

```
tests/grading_wisdom.txt          (12 points)
tests/grading_compareprimes.txt   (12 points)
tests/grading_tellmenow.txt       (12 points)
tests/grading_uniform1.txt        (6 points)
tests/grading_nonuniform1.txt     (12 points)
tests/grading_nonuniform2.txt     (12 points)
tests/grading_nonuniform3.txt     (12 points)
```

For each test, 2 points are awarded for a correct solution (You receive 0 points for the test is your server returns incorrect results). The remaining 10 points (4 in the case of `grading_uniform1.txt`) depend on your server's performance. A detailed explanation of how each test is graded for is output with your score by the test script itself.

The `tests/` directory also has a number of other traces that might be helpful for debugging and testing your server implementation.

Specifically, to emulate our grading configuration, your qsub parameters should be:

```
./run_latedays.sh 4 tests/grading_XXXXXX.txt
```

Hand-in Instructions

In your writeup, please include a table of scores you received when you run each graded test. We will use this to double check your grade when we run your code. Your writeup should include a short (less than one page) description of your scheduling algorithm. (How does the master make scheduling decisions? What is the architecture of the individual workers?)

Name your writeup `writeup.pdf` and make sure all your code is contained in `src/myserver`. Then, `make handin` and submit the resulting tar file on Autolab. This will try to compile your code before creating the tar file, so make sure your code compiles without errors and double-check the output file. If you make a breaking change, you might submit an older version accidentally.

Hints and Tips

1. After getting a correct version of the server up and running, we highly recommend you first focus on getting full performance credit on `tests/grading_wisdom.txt`. It is possible to get full credit using only one worker node.
2. In the `src/asst4include/tools/` directory we have provided you a thread-safe shared work queue (see `work_queue.h`) and an accurate timer (see `cycle_timer.h`). You are not required to use any of this functionality but it may be helpful to you. `WorkQueue::put_work()` adds elements to the queue. `WorkQueue::get_work()` always returns with a valid element from the queue. If the queue is empty, the calling thread blocked until some other thread adds new work to the queue. At this point the blocked thread is resumed and it returns from the call to `get_work()`.
3. There may be opportunities for caching in this assignment. (Hint, take a close look at `tests/grading_uniform1.txt`)
4. Achieving the highest performance will require some thought about the best assignment of jobs to worker nodes. Blindly assigning jobs to thread will not get you full performance credit. You'll definitely need to think about the characteristics of the request processing functions defined in `src/asst4harness/worker/work_engine.cpp`.
5. Parallelism across requests is one obvious axis of parallelism in this assignment. Are there any other opportunities for parallelism?
6. The costs of communication between server nodes is not significant in this assignment. We have specifically designed the client requests to require substantial amounts of processing so as to make communication overheads largely insignificant in the system's overall performance. Put more simply: given how expensive the results are (and given the grading rubic) overheads of scheduling or communicating between the nodes will be negligible.
7. In the past, some students have been unfamiliar with the semantics of **C++ references**.
8. For any C++ questions (including another take that's worth reading on *references*), the **C++ Super-FAQ** is a great resource that explains things in a way that's detailed yet easy to understand (unlike a lot of C++ resources), and was co-written by Bjarne Stroustrup, the creator of C++!
