Designing an Elastic Web Server
DEADLINE 20/12/17

1 Overview

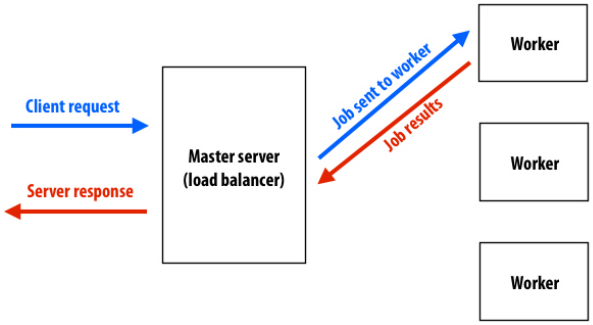
The task of this work is to implement a parallel server that uses a pool of machines to respond to a stream of input requests from a set of clients. Our goal is to respond to all requests as quickly as possible. That is, to minimise the server's response time. However, because running many servers can be costly, our server will have the capability to elastically adapt to variations in request stream load. A good implementation of the server will efficiently take advantage of (i) all the processing resources in a single node (multi-threads), and also use (ii) more machines under times of high load (to minimise response time), and use fewer machines in times of low load (to minimise cost). For now we will focus on the first model, i.e. multi-threaded approach.

2 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 (main thread) 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. For the first part of the project, the master node and each individual worker node in your server configuration will be hosted at the same machine. Hence, a threaded design or an actor model is required.



2.1 The Master

The master node is responsible for interpreting incoming requests and generating the jobs for workers. The master processes the work in an "event-driven" manner. That is, the master process will call the following two important functions when key events happen in the system.

```
handle_client_request(Client client, Request req);
handle_worker_response(Worker worker, Response resp);

 send_client_response(Client client, Response resp);
 send_request_to_worker(Worker worker, Request req);
```

`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: `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` has 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 (`resp.get_tag()` and `resp.get_response()`). The responding worker node is identified by `worker_id` (each worker node is given a unique id which your master learns about in `handle_new_worker_online` – see "Elasticity" section below.).

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.

2.2 The Worker

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

```
worker_node_init(Request params);
worker_handle_request(Request req);
execute_work(Request req, Response resp);
worker_send_response(Response resp);
```

`worker_node_init` is an initialisation 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.

`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`.

2.3 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

```
master_node_init(int max_workers, int tick_period);
handle_tick();
handle_new_worker_online(Worker worker, int tag);
request_new_worker_node(Request req);
```

```
kill_worker_node(Worker worker);
```

`master_node_init` allows to initialise your master implementation. The argument `max_workers` specifies 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 would 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. 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).

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.

2.4 The Incoming Request Stream

To drive your web server, we provide a request generator that plays back traces containing the client ids and the function name to process. For instance, `(1,tellmenow)` is a request coming from the a client of `id =1` and it requires you to process the function named `tellmenow`. The 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 box. However, you'll need to understand the workload created by each type of request (i.e. function to run) to make good scheduling decisions.

418 Oracle. 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.

tellmenow. This request is an automatic "office hours" service for. Someone want help, and they want it now. Hence, this request requires a very strict response latency requirement (150ms). 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 418 oracle. It is CPU intensive with little to no bandwidth or memory footprint requirements. However unlike 418 oracle, the runtime for `countprimes` requests is variable depending on the value of n. Smaller values of n result in cheaper requests.

3 Implementation Details

Following are the main computed to be coded for the project.

1. `Input Module`
2. `Master Node`
3. `Worker` (using the factory design pattern)
4. `Load Balancer`
5. `Client`
6. `Request` and `Response` inherited from a `Message` Class
7. `Output Module`

The `Input Module` will parse a text file containing the mocked client ids and the name of the function to be executed. Each parsed line will mapped to a `Client` and `Request` class. The rested is then fed to an input queue shared with the `Master Node`.

The `Master Node` receives a request try to find the appropriate worker. The assigned worker processes the request and send the response back to the master. There can be a queue between the master the set of workers for such processed requests.

Finally the processed requested are put into the queue of the `Output Module`. The `Worker` implements the aforementioned mocked function, such as `tellmenow`.

The `Load Balancer` is the main class or the module of the systems. It will check the available workers and the chose the workers based on the function to be executed. A good implementation should use the available resources in an efficient manner.

4 How to Submit the Project

All the projects will be submitted through the github account. Push you final project to the github and send me the link through an email (`syed.gillani@insa-lyon.fr`). There should not be any new commits after the deadline.