Atticus Deutsch
CSE130 Asgn3
Document Design
W20

loadbalancer

WHAT IS IT

...

Objective:

This project is an extension to *Asgn2 httpserver* and *Asgn1 httpserver*. Their respective documentation can be found in the *Asgn2 httpserver design document Asgn1 httpserver design document.*

The goal of this program was to create a load balancer to distribute requests to 1 or multiple of the http servers that were created in asgn2

Purpose:

The purpose of a load balancer application is: *Fault tolerance.*

Throughput is only as high as the lowest component throughput of the system, thus whatever the throughput of the load balancer is, no matter how many machines it connects to, we could have just run a multithreaded http server on that same machine without a load balancer and achieved the same requests per second.

Fault tolerance on the other hand is improved with the addition of a load balancer. The load balancer essentially splits the throughput of a maximum multithreaded hw2sever across many machines. That way if one of those machines crashes, throughput doesn't go to 0 as it would if your multithreaded server crashed. If you want to increase your throughput and add a second load balancer, it is important that the two don't over distribute to the same httpserver. This is done by checking the total number or requests and errors of every server periodically in order to find the least busy server or if a server is responsive at all.

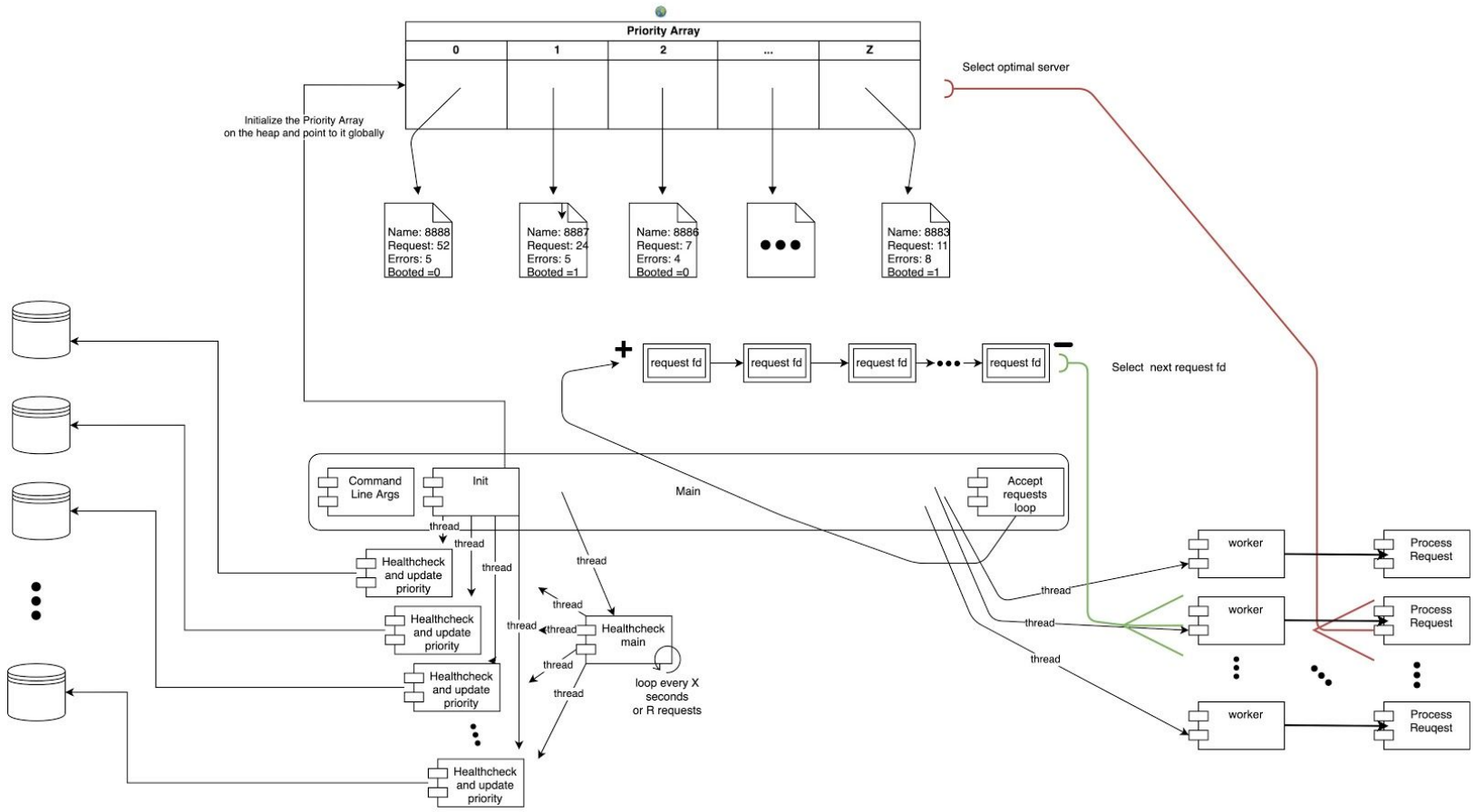
Constraints:

This program will be written in C. This program will not be using any high level http libraries. This program will not support concurrent requests to the same resource.

HOW IT WORKS

…

High level application flow:



Data Structures:

Priority Array:

The priority array is an array of struct hw2server pointers that holds the addresses of the data of each of the servers. Hw2sever struct has a **name,** an **amount of requests,** an **amount of errors,** and a **booted boolean.**

Request Linked List:

Is the same linked list of file descriptors of sockets to communicate with clients that was present in asgn2.

Request Count:

Request count is an integer that represents the amount of **requests - (R*amount of healthchecks/number of servers)**

Modules:

Handle CLA:

Handling the command line arguments is entirely abstracted away into a function called

```
uint8_t handle_cla(int argc, char * argv[], uint16_t * port,
uint16_t ** hw2servers,  uint16_t * num_servers)
```

handle_cla() uses getopt to look for the -N and -R and assigns their values to global variables. Next, the first non-option argument is the port the load balancer will listen on, while the rest of the arguments are the ports where the load balancer can contact the servers. This function makes sure those are valid integers. handle_cla() returns a 0 or a 1. 1 meaning there was an error. After returning, if main sees that handle_cla() returned a 1 it will exit the program with status code 1.

Init:

Initialization of the priority array is done completely in the function:

```
void init_priority(uint16_t * hw2servers)
```

This function will create the priority array on the heap and point to it using a global pointer

```
struct hw2server ** priority_array;
```

Next, this function will create #ofhw2servers threads that all perform healthcheck (see Healthcheck) and then join.

Healthcheck main:

The healthcheck main (dispatcher) is completely executed by the function:

```
void * health_main(void * data)
```

Healthcheck main waits for a signal or 3 seconds to pass using pthread_cond_timedwait().

Next, it subtracts 5 from the request count (or sets the request count to 0 if the request count was already less than 5), and creates #ofhw2servers threads that all perform healthcheck (see Healthcheck).

Healthcheck:

The healthcheck is completely executed by the function:

```
void * healthcheck(void * data)
```

Healthcheck attempts to connect the hw2server passed into its data. If it can't, boot() is called on that server (see Boot) and healthcheck returns.

Next, healthcheck sends a healthcheck request to the server. If 0 bytes were sent then the server is offline so we boot it and return. The same is done if <0 bytes were written.

Next, we time the server's response to the healthcheck using select(). If they time out, then we boot the server. If they respond, then we call handle_healthcheck_response() (see Handle healthcheck response).handle_healthcheck_response() returns -1 then we boot the server

Finally we exit.

Handle Healthcheck response:

Handling the healthcheck response is completely executed by the function:

```
int handle_healthcheck_res(int hw2fd, uint16_t hw2server)
```

This function reads in the health check response until it atleast reads in the whole header.

Next, it checks if the response code is correctly 200 and if there is a content length, if either of those checks fail we return -1.

Next, if there was any payload in that first buffer it copies it into a payload buffer then continues to recv() and copy into the payload buffer as long as there is still content left according to the content length.

If at anypoint the load balancer loses connection to the httpserver it returns -1.

Next, strtok_r() is used to extract the entries and errors from the payload and we make sure they are integers if not returning -1.

Finally, we call update priority() with those two values. (See Update priority).

Update priority:

Updating the priority is completely executed by the function:

```
void update_priority(int hw2server,  uint64_t  requests,  uint64_t
errors)
```

This function iterates through the global priority array and updates this httpserver index with its new number up entries, errors, and the fact that it isn't booted.

Worker main:

As you can see from the diagram after creating the health check dispatches thread the main function creates N worker main threads whose function is called worker_main() and takes in no data:

```
void * worker_main(void * data)
```

Worker_main works in the exact same way as it did in agn2 (please see asgn2 documentation). It will use mutex and condition variable synchronization to take a request fd from the request fd linked list. Then it calls process_request() (See Process request).

Process Request:

Processing the request is completely executed by the function:

```
void process_request(int userfd)
```

First, process_request() calls gethw2sever() (see Get server) which will return the server name with highest priority. If that returned 0 then they were all down and we send a 500 to the user (see sending 500).

Next, we connect with this highest priority server and if that fails we send a 500 to the user and we boot the server .

Next, we facilitate the interaction between the client and server in the same way that the starter code does it, by calling select() in a loop with a 3 second time out. If select() times out, we send a 500 to the user and boot the server. Else we bridge the connections as done in the start code.

Get server:

This function iterates through the global priority array and returns the server name with the highest priority according to the spec. It also increments the request count.

Send 500 to user:

Sending the 500 response to the user is completely executed by the function:

```
void send_500_to_user(int userfd)
```

This function is self explanitory.

Boot:

Booting a server is done with the function:

```
void boot(uint16_t hw2server)
```

This linearly searches through the servers in the priority array for the server with the matching name. It then updates the booted boolean to a value of 1.

Synchronization:

Synchronization was crucial to this assignment since there are maybe shared data structures. In fact, each data structure mentioned in the data structures section has the possibility of different threads trying to access it at the same time.

Priority Array:

The priority array is edited by the boot() function which is called from health check() and process _request(). It is also accessed by update_priority() and by gethw2server(). gethw2server() does not write to the array when looking for the fastest server so we don't need to synchronize it there, but when it updates the request count of the chosen server we need to synchronize using a mutex. boot() and update_priority() do write to the array so they are synchronized using the mutex

Request Linked List:

This is synchronized using a condition variable and a mutex in the same way that it was for asgn2 and described in the asgn2 documentation.

Request Count:

Request count is synchronized using a mutex. I was going to use an atomic integer but I needed to have more than one line be atomic so a mutex is necessary. This makes sure that we know if we have reached the R threshold and requires communication form healthcheck_main() and process request()

Choice of X:

X was chosen to be 3 seconds slightly longer than the chosen timeout time of 2.5 seconds in order for the server to have a half second before we check on it again.