Design Document: Load Balancer

Eric Tang
etang12

1 Goals

The purpose of this assignment is to create a load balancer that is able to communicate with multiple clients and http servers like the one created in the previous assignments and act as a bridge between the two. It will need to work concurrently such that it can handle multiple connections and delegate them to different servers at the same time based on the results of health check calls (from the previous assignment) to all the servers. Effectively, the load balancer will act as the medium in which a client and server can interact with another; the load balancer will simultaneously act as a server listening for client connections and as a client sending requests to an http server concurrently while also sending http server responses back to the client.

2 Design and Implementation

2.0 Design Flow For my design, I took note that this assignment would not work without a fully functioning multithreaded http server with logging enabled from the last assignment. Luckily, the teaching staff decided to provide us with a binary http server so I didn't have to worry about heisenbugs that I may or may not have had with my own implementation of the http server. With that in mind, I only needed to really focus on the aspect of a load balancer and how it worked. Like the previous assignments, I first attempted to figure out how I would separate the assignment into smaller subsections to deal with. The first thing I realized was that the load balancer was essentially just a tool that would act as a client AND a server in the sense that it would listen for incoming client connections like a server, and send client requests to a http server just like a client. So, I decided that I would use my implementation of multithreading from the previous assignment in this one so that I would be able to concurrently store and handle client connections by using a shared queue in a dispatcher/worker thread method. This way, I knew I would be able to store client connections inside of a queue that I could use to send to http servers at a later point in the assignment. I made sure that I could get the load balancer to act as a server before moving onto the next thing. In this case, the next thing would be getting the load balancer to send those requests that I had stored in a queue to different http servers. The problem was figuring out how I would know which server to send requests to as well as how I would access those servers. I eventually ended up with the idea that I would store all the servers inside

of an array filled with structs. This struct would contain information such as server port number, how many entries and errors the server currently has logged, and other indicators that would help determine which server would have priority over others in receiving requests. The server priority is decided by how many total entries that the server has serviced. If multiple servers share the same total number of serviced requests, then the server with the higher success rate (less errors) would be chosen. If they all have the same success rate as well, then any arbitrary server will be chosen. So, I would need to find some way to use this array of server structs in conjunction with the queue of client connections in order to make my load balancer function as both a client and server. I also decided to use the starter code as it allowed me to understand how the load balancer connected clients to http servers.

2.1.0 Main Function My main function works very similarly to my implementation of it from assignment 2 such that it acts as a dispatcher thread that enqueues client connections into a shared queue with some slight modifications to consider a separate health check thread. Essentially, this function parses the command line to save necessary information about how the load balancer should run, including N parallel connections that can be handled concurrently, R requests that should occur before sending another health check to all the servers, the port number that the load balancer should act as a server and listen for connections, and finally the port numbers of the http servers. The command line is handled with the getopt() function just like last assignment. After parsing the command line for the essential information, I needed to initialize the server structs using the port numbers specified by calling the function init_servers(). From there, I spun up my health check thread and made it call health check every X seconds or after every R requests to match the specification. The information from the health check is stored back into the server array and to their respective servers. Everything after involves spinning up the worker threads to handle N connections in parallel and having the dispatcher enqueue client connections to the queue. Also make sure to include mutexes to ensure concurrency and prevent threads from stepping into each other's memory and creating issues/race conditions.

```
int main(argc, argv) {
        use getopt() to parse command line and store information appropriately
        call init_server() to initialize server struct array
        spin up health check thread and call initial health check to servers
        create shared queue
        spin up worker threads to handle N connections concurrently
        enqueue client connections to the shared queue
}
```

2.1.1 init_server An initializing function that is only run once to allocate space for an array of server structs. This function stores the server port numbers specified on the command line into the array as well.

2.1.2 server struct This struct contains all the fields that determine which server gets priority of receiving client connections from the load balancer. Used by being stored in an array that gets updated from every health check request to all the servers.
The fields contained in this struct are:

1. port number (server port)
2. number of entries logged (how many requests the server has serviced total)
3. number of errors logged (how many requests that failed)
4. status code (used to check if server returns 200 as a status code, else server is down)
5. healthcheck_len (stores the content length of the health check request, used for parsing)
6. success_rate (used as tie breaker if servers have same number of entries)
7. min_index (used by worker thread to choose which server to send client connection to)
8. alive (boolean that determines if a server is down or up)

2.2.0 client_connect Function provided with the starter code. Uses port number from server array to connect load balancer to the correct http server. Returns a socket that is connected to a server. Allows the load balancer to act as a client to the server.

2.2.1 server_listen Function provided with the starter code. Uses port number from command line to act as a listening port for the load balancer. Load balancer then uses the port number to listen for client connections, accepts them, and stores the connection in the queue through the dispatcher thread. Allows the load balancer to act as a server to client connections.

2.2.2 bridge_loop Function provided with the starter code. This function takes in a server port file descriptor and a client connection file descriptor. It allows the load balancer to forward messages from one file descriptor to the other depending if there is data to be sent. The select() function is useful because it allows the program to check if a file descriptor has bytes in it. It determines which file descriptor has bytes to read/write so that I can determine when a client is ready to send data to the server, and vice versa. This way a client is allowed to send data to a server then the server's response is sent back to the client in the correct order. Select() determines which stage the data sending is in (if client → server, or server → client). Another function called bridge_connections() is called in this function to actually send data to and from these socket file descriptors.

bridge_loop(sockfd1, sockfd2) {

infinite while loop {
select() to see if sockfd1 or sockfd2 are ready (have data to be read or sent)
if select returns -1 then error during select
if select returns 0 then server has timed out, didn't send response to load balancer
default: if sockfd1 has data to be sent, then mark it as fd with data (fromfd) else if sockfd2 has data to be then mark it as fd with data (fromfd)
the sockfd that isn't marked with data will be marked as fd to receive data (tofd)
}
call bridge_connections(fromfd, tofd) on the fd marked to send data and on the fd marked to receive data
}

2.2.3 bridge_connections Function provided with the starter code. This function handles the actual sending and receiving of data between the two socket file descriptors in the bridge_loop() function. It takes the two file descriptors from that function as input, however it can differentiate between the two file descriptors and know which one is ready to send data (fromfd) and which is ready to receive data (tofd). So, it receives (using recv()) the data from fromfd into a buffer, then sends (using send()) it to tofd to receive data.

```
bridge_connections(fromfd, tofd) {
    recv() data from fromfd
    send() data to tofd
}
```

2.3.0 healthcheck_func A big portion of this program depends on sending health checks properly to all the http servers. This code effectively loops through the array of servers initialized earlier and sends a health check request to each one. It then receives (recv()) the response from the server and parses it using sscanf(). The parsed responses from the servers are stored inside of the server struct at that current index of the loop. After storing the parsed response, the function continues by using another loop to iterate through the entire array of servers. However, this time it is used to compare each server with each other to determine which server should be the most prioritized. This is achieved by setting the first server in the array as the "min". From there, I compare the entries from the min server with the next server in the array. If the next server has less entries then it becomes the new min. If they have the same number of entries, then the success rate of the servers is calculated (1 - (errors / entries)), and the server with the larger success rate will become the new min. If the servers both have the same entries and success rate then the server compared with the current min gets set to min. It iterates through the entire server array and compares the current min with the next server and updates the min when necessary.

The min is then stored in the min_index of the server struct, and used by other functions that rely on using a server port.

```
healthcheck_func() {
        for loop to iterate through entire server array using index i {
                connect to server port at index i
                dprintf() and send health check to server
                recv() response from server
                sscanf() the response and parse it into server struct
        }
        set min = first server in array
        for loop to iterate through entire server array using index i {
                if server at current index i has entries less than min, then update min
                else if server at current index i has entries equal to min, then check success rate
                        if success rate is higher than min, update min
                        else set min to be server at current index i
        }
}
```