Design Document for Assignment 3

Matthew Klein

Goal

The goal of this program is to build a load balancer that handles requests from clients, forwards them to an instance of the httpserver, and sends a response back to the client. The load balancer does not parse or read any data except for that from the healthcheck, and merely sends along each request from a client to a running server. Which instance the request is sent to is determined by running periodic healthchecks to each instance and determining the priority in which the next request should be sent. Once the insatnce of the server has processed the request, the load balancer forwards the response from the server to the client, along with any data.

Design Specifics

The first thing to accomplish will be to parse the program arguments. As with the actual server, the arguments will be parsed using `getopt()`. This allows us to determine the value of certain parameters if they are specified within the program args. The port number in which the load balancer, abbreviated as *LB* from now on, will always be the first required argument after the optional ones parsed using `getopt()`. All numbers specified in the args must be positive, or we exit the program. We will store the server ports in a linked list to be able to access them. Each will store its own local variables that contain information about the specfic server. This includes the server port number, the status of the healthcheck, the number of requests processed, and the success rate for the server.

We then run the initial healtthcheck on each of the servers and update their status and variables accordingly. Each node in the list will also contain its place within the list in order to understand the dependencies.

To store the client socket FDs that send a request to the LB, a circular buffer will be used. This *CB* will shared between the threads as the thread will dequeue this data structure will be copied from asgn2 as queuing and removing client socket FDs worked from the last asssignment. We loop accept until the LB does not have any more client sockets that need a request processed. To store the server ports and other info about them, a linked list will be used.

A struct of thread args will be ceated to store the shared CB and the linked list that contains the server ports. By creating *N* threads for the LB to process requests, we can handle multiple requests from clients simultaneously without increasing complexity immensely. The thread arg will contain a circular buffer pointer to allow each thread to dequeue awaiting clients as necessary. It will also contain a pointer to the beginning of the linked list storing the server ports.

The main thread will dispatch worker threads to complete the work that will be assigned to them from the dequeued socket within the CB. The worker thread will handle dequeue the same as it did in asgn2, by obtaining the lock of the CB. Once it has this client socket, it will search through the list of servers to choose which one will be

used, based on its priority. It will then create a connection with the server, and then call the provided `bridge_loop()` function to forward data between the sockets.

Health checking will be completed in its own thread. For every 4 seconds, or *R* requests given in program args, the healthcheck will run again and update the priority of each server in the list.

Algorithms/Logic

As with the parsing of program arguments from asgn2, we will use `getopt()` and switch cases within a loop, in order to fully parse the program arguments.

```
while(getopt != -1){
    switch{
        case N:
            ...
        case R:
            ...
        default:
            ...
    }
}
```

R is the number of requests the server will process before the LB runs another healthcheck on an instance. *N* is the number of concurrent requests the LB can handle. After parsing optionals, we will then loop through the rest of argv[], as it will contain the other arguments, mainly the port number that the LB will accept new client requests. Each server port will be stored in a variable array, as we do not know the number of servers we will connect to, until run time. The variable array allows us to dynmically allocate space to store the server port numbers as they are parsed from the rest of the program args.

In order to store the server ports, a linked list of nodes will be used to store the server ports and other information about each server, such as priority, status of the server, and the data of the healthcheck.

```
struct Node{
  int serv_port;
  int errs;
  int priority;
  int entries;
  int status;
  int success_rate;
  char *serv1_port;
  struct Node *next;
};
```

Each node will contain its own information so the whole list can be passed between differnt functions. The server port will be passed in from the parsed args, and will be converted to and stored as `serv_port`. The rest of the variables, except for `next`, will be updated in the healthchecking function. A simple insert function will also be needed to create the linked list upon receiving and parsing the ports

specified in the program args. The ThreadArg, which will be passed into the threads, will contain a pointer to the head.

After initializing everything needed for the CB and ThreadArg, we will create the `HC_thread`. This thread will handle the complete functionality of the helathchecking for the program. It will take in a ThreadArg to be able to obtain the pointer to the head of the linked list. Inside the thread, there will be a `pthread_cond_timedwait()`. This will allow the HC thread to wait a chosen amount of time, or until a specified, *R*, requests have been processed by the LB. This will be handled by locks and signaling condition variables to release said locks. Inside the while loop, the HC function will be called. The function handles all of the functionality of the healthchecking. The first thing to do, is to make sure we loop through the entire list and run a healthcheck on each server. This can be accomplished through a simple for loop,

```
struct Node *ptr;
for(ptr = list; ptr != NULL; ptr = ptr->next){
    ...
}
```

Inside the for loop we will first open a connection to the server from the LB. This will allow us to send and receive data. After connecting to the server we will use `dprintf()` to send a GET request on healthcheck, to the server. After, we will use `recv()` to recv into the buffer. `sscanf()` will be used to parse the needed information from the recv calls. We then will determine if the the status code of the response is 200. If it is, we can continue to parse more information about the healthcheck. From this parse, we determine the success rate of each server, the total entires, and the number of errors. Othwerwise, if the status code is not 200, we mark the server as down.

```
dprintf(connfd, "GET /healthcheck HTTP/1.1\r\n\r\n");
recv_all();
strstr(200);
sscanf(recv_buffer);
if(status_code == 200){
    sscanf(errors, entires);
    ptr->success_rate = 1 - (errors/entries) //if entires != 0
    min = ptr->min_ents
}
else{
    ptr->status = 0
    ptr->priority = 0;
}
```

Then we will loop through again to do more comaprisons and correctly determine the server which will have the highest priority.

```
for(x:list){
    if (x is running)
        if (min->entires == x->entires)
            if(min->successrate >= x->successrate)
                continue;
            else
```

```
                min = x;
}
min->priority = 1;
```

This functionality will continue for the entire life of the server and will run every *X* seconds, and for every *R* requests.

Next we will create a set of N threads, where *N* is specified within the program args. This will allow the LB to handle concurrent requests up to a number *N*. Once inside the thread, we will wait for a signal from the enqueue function before we can progress. Once we have the lock, we will dequeue a client socket waiting, and store that in a local var. We can then release the lock and signal the enqueue to allow more clients to be added. A check on servers must then be preformed to ensure that the server with first priority is still running. If the server is still up, we can set up a connection to it by calling `client_connect()` on the port number of the server. The client_connect function is taken from the starter code and not modified. Once a connection is established, `bridge_loop(client socket, server socket)`. Total requests are updated to be used in the healthcheck thread, and we close the sockets at the end of the loop.

```
while(1){
    obatin client socket
    check if server is up and connect ot server
    bridge_loop(client socket, server socket);
    clsoe (client & server sockets)
}
```

The `bridge_loop()` and `bridge_connections()` functions will only be slightly modified to meet the needs of the specification. Bridge loop will only modify the 0 and -1 cases of the switch statment within. The function will send a 500 error and close the sockets

```
dprintf(client, "HTTP/1.1 500 Internal Server Error\r\nContent-Length: 0\r\n\r\n");
```

After testing, the `bridge_connections()` needed to be modified. The program was not returning correctly and editing this function resloved the issue. All `printf()` statements were removed from the function, the initial `sleep()` was removed, and the receiving buffer was cleared at the beginning of each call. The buffer needed to be cleared as it was no being fully overwritten on subsequent calls to the function.

```
memset(recvline, 0, strlen(recvline));
```

As with assignment 2, I implemented a `while(1)` loop to continuously accept client connections, into a circular buffer, for the life of the program. The loop would wait for a signal from a dequeue, if it was full, to start another enqueue.

```
while(1){
    pthread lock
    if(queue is full)
        wait
    call enqueue
    pthread unlock
```

```
    pthread cond signal
}
```

Testing

Testing will be done in two stages initially. The first will be testing the forwarding functionality of the server. To do so, manual curl requests will be sent to the loadbalancer port using the curl comamd.

```
$ curl localhost:1234/test
```

To test concurrent requests, multiple curl commands can be sent simultaneously

```
$ curl localhost:1234/test & curl -I localhost:1234/test
```

Next would be to test the LB's ability to properly healthcheck. The first tests will just have the httpserver running with -L enabled to make sure healthchecking is on to make sure the server(s) can return the intial healthcheck properly. Then more testing will be done by sending requests to the LB and seeing how often a helath check runs.