DESIGN DOC for loadbalancer.c (asgn3 CSE130)

Author: Adarsh Sekar

1.Introduction:

1.1 Goals and Objectives:

The goal of this assignment is to create a load balancer to balance the number of client http requests amongst multiple servers. This will allow us to send client requests to a single location/server, which will in turn send it to multiple servers whose ports it has access to. An overview is that we are essentially performing something similar to multithreading, but instead of threads, we have multiple servers that themselves can have multiple threads. This is done so that no single server is overburdened, and to increase efficiency in the case of multiple http requests, by spreading the load of the requests amongst multiple servers.

1.2 Statement of Scope:

This program will be called with a port number used to receive http requests, followed by port numbers used by the servers, where at least one server port number must be provided. The program can also be called with optional parameters -N, signifying the number of parallel connections, and -R, signifying the number of requests after which we should call a health check. For the health check, I will also be choosing a variable X, that should be less than or equal to 5. I will choose X to be 2. I will be calling a health check after either R requests or X seconds, whichever comes first. Below are examples of how the code can be run:

```
./loadbalancer 1234 8080 8081 8743
./loadbalancer -N 7 1234 8080 -R 3 8081
./loadbalancer 1234 -N 8 8081
./loadbalancer 1234 9009 -R 12 8080
./loadbalancer 6570 1235 -R 2 1236 -N 11
```

1.2.1 Selection of X:

The reason I selected X = 2, is because when I was running httpserver in asgn2 for files around 400MiB, I noticed that it took around 0.1 seconds per http request. So with 2 seconds, that is around 20 requests. This is a good amount such that we aren't calling health check too often that we are creating a bottleneck for our program, but also won't allow too many requests to be parsed before we switch to a different server, allowing the servers to stay relatively balanced. Even if the files are less than 400MiB, an X of 2 will limit the number of requests as there is still a minimum amount of time that the httpserver takes to parse any request. Therefore, no server will be overloaded with a value like this.

2. Data and Code Design:

In this section, I will go over my design for my data structures and code.The design for this assignment is fairly simple. I will not go over data types that I explained in previous assignments. This section will be split into 4 parts: Data Types, Helper Functions, Main functions and main().

2.1 Data Types:

```
struct workerThread{
   pthread_t worker_id;
   int id;
   pthread_mutex_t* lock; //Lock to get cfd
   pthread_mutex_t* dlock;
   pthread_mutex_t* rlock;//Lock to increment the number of requests
   pthread_mutex_t* hlock; //lock for healthcheck
   uint16_t* numreq; //Pointer to the number of requests
   uint16_t recheck; //Number of requests at which we call a healthcheck
   uint16_t* portarr; //Array storing all the server ports
   uint16_t numports; //Gives the number of ports, as it may nto equal array length
   uint64_t* serversCount; //This will be a pointer to a 2D array of servers, where each index will refer to the count of the requests to the servers
   int cfd;
   int* free;
   pthread_cond_t cond_var;
   pthread_cond_t* healthsig; //Condition variable used to signal the health check thread to wake up
   pthread_cond_t* workersig; //Condition variable I will use for the health check to signal the worker threads that it is done with the healthcheck
   pthread_cond_t* dissignal;
   bool* healthcheck; //A pointer to a bool so that worker threads can check if there is a health check currently happening, and wait for it to be over before continuing to process requests
};
```

This struct will be used to store the data of the worker threads. We will have a mutex lock for the getting a valid cfd, and another one when we want to increment the number of requests, a uint16_t pointer to the number of requests (numreq), an int pointer to the server socket descriptor (server), a uint16_t index for the index of the thread in our array, and a boolean pointer free, that points to a variable that signifies whether the thread is free or not. I also will have an int for the client socket descriptor.

```
struct healthThread{
   pthread_t health_id;
   pthread_mutex_t* hlock; //lock for the health thread
   pthread_cond_t* healthsig; //condition variable we will use to signal the health thread
   pthread_cond_t* workersig; //Condition variable we will use to signal the worker thread to wake up.
   uint64_t* servers;
   uint16_t* portarr; //Array storing all the server ports
   uint16_t numports; //Gives the number of ports, as it may nto equal array length
   uint16_t* numreq; //Pointer to the number of requests so we can reset it
```

```
    bool* healthcheck; //bool we have a pointer to to let worker threads know there is a
healthcheck going on

};
```

This struct is for the health check thread, as it needs different information than the worker thread. We have the thread health_id. A mutex lock for when we are using pthread_cond_wait and pthread_cond_signal. Two condition variables, healthsig is the one we wait on, and workersig is to signal the worker thread to wake up. A pointer to the server, a pointer to an array of ports. Then I have the number of ports, and a pointer to the number of requests, so I can reset it.

2.2 Helper Functions:

`int bestServer(uint16_t* ports):`

This function will be used to let the different load balancer threads know which server port to connect to. It takes in a pointer to a 2 dimensional array ports, and returns the index of the best port.

This function will go through every index of our array, and return the index with the least number of requests. If 2 indexes have the same number of requests, we return the one with the least amount of errors. If two or more indexes have the same number of requests and errors, I will return the one I first saw. If the last element (index 2), of any of the subarray is 1, that means the server failed a healthcheck, and so should not be used.

`int bridge_loop(int sockfd1, int sockfd2):`

This function is mostly the same as it was in the starter code, except I return -1 in the case of failures, and 0 if nothing bad happened. I have two booleans that I initialize to false before the while loop, called conngoing, conncoming. I set conncoming to true if the client is sending data to the server, and conngoing to true when the server is sending data to the client. If select returns 0, then I if conngoing or conncoming are false, I send a 500 error to the client. If conncoming is true, then I see return if bridge_connections returns 0, and continue waiting otherwise. Otherwise, I return -1;

2.3 Main Functions:

`void* worker(void* data):`

This function will be used for all the worker threads. It will be where we have a load balancer make client connections with our httpservers. Now what we want is for every thread to make a connection with the appropriate server, send it the client request, and receive the response from the server. The way I will do this is by creating a while(true) loop inside worker().

As soon as I enter the worker function, I will first create a workerThread() pointer ‘this_thread’, and store my void* data into this variable. Then I will enter the while loop, and immediately store the server port number in a local variable. The reason for this is in case of a health check calls, the port number should not be changed midway through a client server connection. Inside the while loop, we'll lock our start mutex. Then, I will check if *this_thread->numreq mod this_thread->recheck is 0. If it is, I set this_thread->healthcheck to true, and signal the healthcheck thread. Then I put the thread to sleep if this_thread->healthcheck equals true inside of a while loop.

When the thread wakes up, I set this_thread->free to this_thread->id. I then wait until Iget a client socket descriptor. The current thread will sleep until it gets the client socket descriptor. When the value of this_thread->cfd is positive, it means we got the client. Our thread would be signalled to wake up. Once it does, I get the best server to use, using my function bestServer(), and store it in a local int index. If index is less than 0, I send a 500 error to the client, and close the client, unlock the mutex, and continue through the loop. Otherwise, I set local variable server to client_connect when called on this_thread->portarr[index]. I increment the count for that server using this_thread->serversCount, and also increment this_thread->numreq, and then unlock the mutex.

After we get our client, we connect to our server port using client_connect(). Once we connect to the port, I call bridge_loop() between our two socket descriptors, storing the return value in an int variable called bad. I then lock rlock, and check if bad is less than 0. If it is, there was a problem sending data for that thread, so I increment the number of errors for that thread using this_thread->serversCount.

void* healthcheck(void* data):

I will have a separate thread for health checks, that will perform health checks continuously, and change the server port appropriately. As soon as we enter the function, just like in worker(), I will store the data in a local variable health_thread. I will also create local variables timespec ts and timeval curr.Then, we will enter a while(true) loop.

Inside the while loop, I will first get the time using gettimenow, and storing the time in curr. Then I will set ts's seconds to curr.tv_sec + 2. Then I will put the thread to sleep using pthread_cond_timedwait() on ts. Once the thread wakes up, I will send a health check request to each of our ports, and store the number of errors and number of requests in health_thread->servers. If a port is unresponsive, we will mark it as such, by setting it to 1. Once I'm done checking all threads, I will set health_thread->healthcheck to false, and signal the worker thread.

2.4 main(int argc, char** argv):

My main() function will be pretty simple, and somewhat similar to the main() function in asgn2. The overview is that we will parse in the loadbalancer arguments using getopt(), and then create a number of threads N to run multiple load balancers in parallel. This time, I will also have a thread that will be used for health checks, that I will explain later in this section.

The first thing that I will do is initialize some variables. I will need an uint64_t array to store the different ports, that will be of size argc - 2 (because argc includes the program itself as an argument, and also we subtract the load balancers port), and an int to keep track of how many port numbers were provided, as this may be less than the size of my array. I will also create int variables to store the value any -N and -R options. These variables will be initialised to 4 and -1 respectively. Now, I will go into the getopt() while loop, and store the port numbers into my array whenever I see them, and optionally a -N and -R variable if I see them. If I see any other headers, I will return EXIT_FAILURE.

Now that I have gotten all the information I need using getopt, I will create the appropriate number of threads of my load balancer given by the value of N, and then create an extra thread for the healthcheck.

After I create the respective threads, I will enter a while loop where I receive clients and send them to the load balancer threads, where they will in turn be sending the clients to servers.