Load Balancer Design

Jake Armendariz

Goal

The goal is to create a loadbalancer that will act as single port, but will forward http requests to 1 or more multi threaded httpservers, which will create and send response to a loadbalancer, who will forward the response to a client. This should increase throughput of threads by having multiple servers handling requests. Sadly, it will increase overhead, as it takes more time for a request to make it to a server, and back to client. In order to keep track of not only which servers are up and down, but which server is most likely to be up. We send healthchecks to server every 3 seconds in order to prioritize requests going through load balancer.

Analogy

While preparing for Christmas Santa will have to build thousands of presents. But he can't do it alone, he needs his elves, which are seperated into groups of workers across the north pole. For every boy and girl santa sends a custom toy build request to each group of eleves. Sometimes theres a fire, or equipment breaks and these groups cannot make toys, so santa will give them a break (while that group puts out the fire) and santa will only send toys to elf groups that can handle more toy requests. In order to know which elven groups are functional, Santa has his Mrs. Claus check in with all the groups alot, to tell him who is ready for more work, and which groups need breaks. After recieving built toys from the elves, santa delivers the package to a special cubby for every boy and girl.

- Santa is the load balancer
- Present requests are http requests
- Each elf group is a multithreaded http server
- Completed toys are the http response
- Each boy and girl, along with their cubbies, are clients.
- Mrs. Claus is a healthcheck thread, responsible for telling loadbalancer which servers are running

Design:

Once load balencer is called, it will create a list of server-info structs, inside of my Servers struct. Both of these structures are based off of Clarks section on May 28th. Using getopt I will initialize my enviorment, including the R and N. Then I will create the list from the remainining servers.

Each httpserver is idenitfied by a serverInfo struct, that contains its port number, an bool value indicating if its alive. And the total-requests and error-requests are set equal to each healthcheck update. I chose to update the total-requests for everytime a server is chosen, but reinitialized by healthcheck. This is attempting to keep the healthchecks up to date when a server is handling multiple large files and cannot respond to healthchecks in time.

```
struct ServerInfo {
    int port;
    int fd;
    bool alive;
    int total_requests;
    int error_requests;
};
```

Servers will contain an array of ServerInfo to indiciate each httpserver. The other variables allow me to create the enviorment and pass it to my functions.

```
struct Servers {
    int num_servers;
    struct ServerInfo *servers;
    pthread_mutex_t mut;
    int num_threads;
    int request_count;
    int R;
    int N;
};
```

Going to start off single threaded, sending each request to the best server, using serverfd and select to make the decison.

Dealing with Connections

Load balancer is instatiated as a listening port, and each request is taken in with accept(), I save

the file descriptor, and then choose which port addr should handle the request. To connect with the file descriptor, I use client_connect() to connect to existing http server, if this fails, I will automatically send a 500 INTERNAL SERVER ERROR response. If successful, the load balancer file descriptor that accepted request, and the server file descriptor to handle it will both be sent to my threaded function `void *process_data(void *data)` which will forward the data to server, and back to load balancer. Then, after the response was handled, both file descriptors are closed, and the thread is killed.

Connection Forwarding

Each request is accepted by the loadbalancer port. Then, based off of total-requests and error-requests I am following Clarks suggestion of sending to alive port with the most error-requests because its most likely not to be processing data due to nature of error responses being faster. After finding the best port, create a connection, pass a pthread into a function to send, process and recieve the request from server. To limit the number of pthreads working at once I set a counter that will increase when starting, decrease once finished and my main thread will wait on a condition vairbales for the number of running threads to be < N.

Handeling large requests
For large get and put requests of binary data we need to rely on the message-header -> content-length to read in all of the data. To do this, I have a function that parses out the content length from every request sent to my server. I will forward each buffer to the httpserver, and will continue to read/write as I send to server.

Health Checks

I have a single thread, mutex and condition variable just for healthchecks. It will loop through waiting for 3 seconds with p-cond-wait or R requests, then will signal and enumerate through the servers, sending and recieving healthchecks one at a time. If a server doesn't respond, it will be marked with server->alive = false;

To parse each healthcheck, I check to make sure it contains 200 OK, if not, server is down. Then I parse out the total-requests and error-requests from the healthcheck, updating the ServerInfo struct in each array. Unsure if the body will be sent with header, I look for the double carriage, if its index < bytes recieved -4, then I know the body was included in buffer and parse from that. Otherwise, I call recv again to get healthcheck body.

```
void* healthchecks(void *data){
    struct Servers *servers = (struct Servers *)data;
    struct timespec ts;
    struct timeval now;

    while(true){
        char * healthcheck = "GET /healthcheck HTTP/1.1\r\nContent-Length:
        for(int i = 1; i < servers->num-servers; i++){
            if((serverfd = client-connect(servers->servers[i].port)) < 0){
                servers->servers[i].alive = false;
                close(serverfd);
                continue;
            }
            if(send(serverfd, healthcheck, strlen(healthcheck), 0) < 0){
                servers->servers[i].alive = false;
                close(serverfd);
                continue;
            }
            char buff[255];
            int n = recv(serverfd, buff, 255, 0);
            if(buff is not a valid response){
                   servers->servers[i].alive = false;
                close(serverfd);
                continue;
            }
            total, error = extract-data(buff);

            servers->servers[i].total-requests = total;
            servers->servers[i].error-requests = err;
            servers->servers[i].alive = true;
            close(serverfd);
        }
        pthread-mutex-lock(&hc-mutex);
        memset(&ts, 0, sizeof(ts));

        gettimeofday(&now, NULL);
        ts.tv-sec = now.tv-sec + 3;
        pthread-cond-timedwait(&hc-cond, &hc-mutex, &ts);

        pthread-mutex-unlock(&hc-mutex);
    }
}
```

Not much else to cover, fairly intuitive once you understand the spec.