Load Balancer Design Document

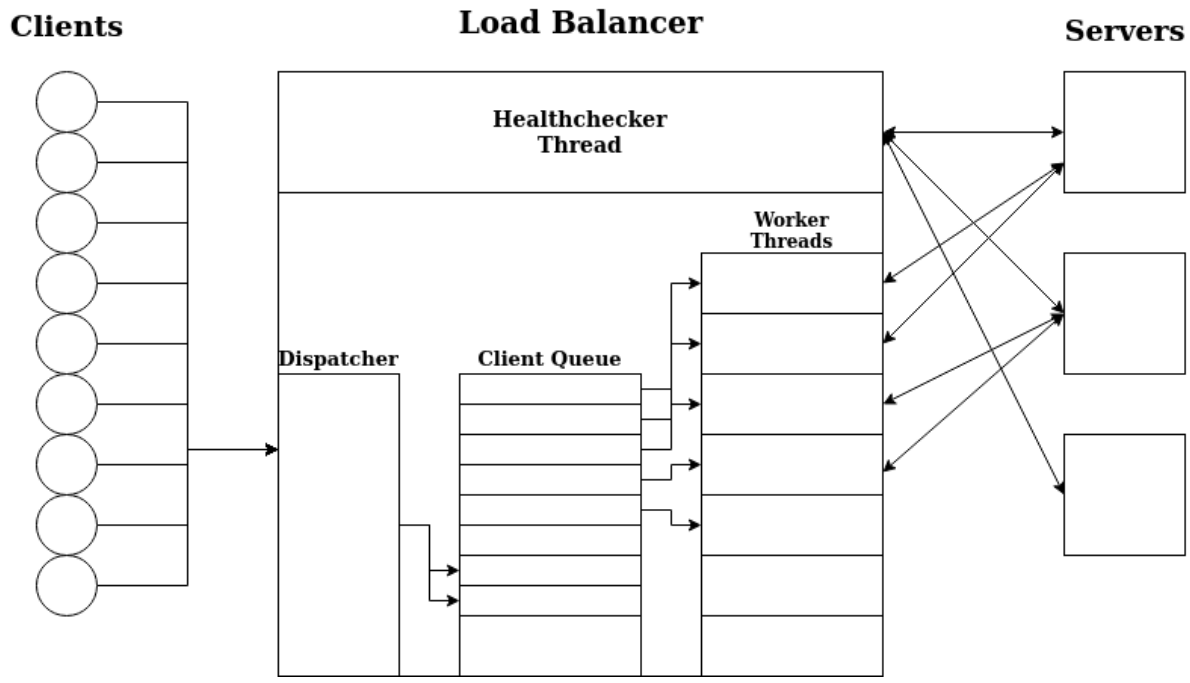
Author: Austin Seyboldt

Design Overview

My program will have a main dispatcher thread which listens for incoming connections from clients. Once a connection arrives, it will be accepted and the client will be added to a queue. From that point, any one of the available N worker threads can dequeue the client, connect to a server, and begin forwarding the connection. In addition, there will be a healthchecker thread, which waits until either R requests have been processed or X seconds have passed, at which point it will wake up and request healthchecks from all the servers. This will be performed in parallel by having a separate thread for communicating with each server. This is so that if a server fails to respond, the healthcheck can still receive a response from the other servers in a timely manner.

My design is operating under the assumption that a sufficient largely N (approx equal to the total number of worker threads across the http servers) will be specified, otherwise the load balancer will not be able to take full advantage of the multithreading of the servers.

Notice that the healthchecker thread will be asynchronous with regards to the worker threads, making it possible for the worker threads to accept new connections while the healthcheck is being performed. Thus, more than R requests could be processed before the healthcheck has finished and updated the shared data structure. The spec does not seems to explicitly mandate synchrony here, so I have chosen this implementation to favor improved concurrency/throughput. My healthcheck will instead probe for a healthcheck every R requests or X seconds *since it last finished a previous healthcheck.*



Specific Design choices:

I chose to make my healthcheck timeout be 1 sec after trying different values because I wanted to ensure that if the server didn't respond quickly, it should be marked as down because it is probably exceeding its capacity. I made the time between healthchecks .8 seconds because I wanted it to be quick enough that my loadbalancer had a near "realtime" idea of the state of the servers and also so that if a worker thread marks a server as bad, the healtheck can check again if its back in operation soon.

The size of my buffer for fowarding data is 8192 bytes, which is large enough to transfer large amounts of data quickly. I suppose I could've made it even larger, but based on Anonymous Poet's post on Piazza (@344), the difference in speed for different buffer sizes was neglible for buffers between 8K bytes and 64k bytes.

Data that must be shared:

The worker threads and the healthcheck thread must share a data structure for tracking the available servers. They will use an array. The worker threads need to access this to figure out which server a new connection should be routed to, as well as incrementing the request count of a server when sending a new request to it. The healthcheck will need to update the array whenever it performs a healthcheck, changing the request totals to their true values and marking any unresponsive servers.

Additionally, the main thread which performs initilization of all data structures, mutexes, etc will need to share a master list (an array) of all servers with the healthcheck thread. This is so that the healthcheck thread will know about all servers (since the heap will only track working servers).

Data Structures

A structure for storing information about a server will be needed:

```
struct server_info {
    uint16_t port
    size_t total_requests
    size_t num_failures
    char address []
    bool is_down
}
```

There will also need to be data structures for passing data into the threads.

```
struct worker_args {
    pthread_cond_t* wake_healthcheck    // used to signal healthcheck to wakeup
    pthread_mutex_t* dispatch_lock        // used to lock access to client queue
    pthread_mutex_t* server_array_lock    // used to lock access to array of servers
    pthread_mutex_t* health_request_update_lock        // lock access to requests_since_healthcheck
    pthread_cond_t* client_ready        //

    size_t requests_before_healthcheck    // num requests before performing healthcheck
    size_t* requests_since_healthcheck    // tracks num requests since last healthcheck
    server_info* server_array             // array of info of all servers
    size_t server_count                    // number of servers
    queue* client_queue                 // used to grab new clients to serve
}



struct healthcheck_args {
    pthread_cond_t* wake_healthcheck     // used to receive signal from worker thread to do healthcheck
    sem_t* health_init                    // used to make the main thread wait for initialization
    pthread_mutex_t* server_array_lock    // lock access to server array
    pthread_mutex_t* health_request_update_lock    // lock access to requests_since_healthcheck

    size_t requests_before_healthcheck    // num of requests before doing a healthcheck
    size_t* requests_since_healthcheck    // num requests since last healthcheck
    server_info* server_array             // array of info of all servers
    size_t serevr_count                 // num of servers
}



struct healthcheck_worker_args {
    server_info* serv_info                    // info about this worker's server (will be updated)
    pthread_mutex_t* worker_wake_lock;        // used to check worker_wake_count
    pthread_cond_t* worker_wake;             // used to wake up the worker
```

```
    pthread_barrier_t* worker_barrier;        // used to sync all workers/main health thread
    size_t* worker_wake_count;                // count of awake threads
    size_t thread_id;
}
```

Modules / Functions

The dispatcher thread simply accepts client requests and adds them to a queue to be processed by the worker threads.

```
dispatcher thread:
    while true:
        new_client = accept(server_sock)
        pthread_mutex_lock(client_queue_lock)
        client_queue.enqueue(new_client)
        pthread_cond_signal(client_ready, client_queue_lock)
        pthread_mutex_unlock(client_queue_lock)
```

The worker thread grabs a client from the client queue and calls a function to choose the optimal server from the server_array. It also needs increment a count of how many requests have been received since the last healthcheck and if its equal to R, it must signal the healthcheck thread. The worker then begins bridging the connection between the client and server.

```
worker thread:
    while true:
        // receieve a client connection
        pthread_mutex_lock(client_queue_lock)
        while (client_queue.empty())
            pthread_cond_wait(client_ready, client_queue_lock)
        client_fd = client_queue.dequeue()
        pthread_mutex_unlock(client_queue_lock)

        // get server info of optimal server
        pthread_mutex_lock(server_array_lock)
        server_info* serv = get_optimal_server(server_array, server_count)
        if (serv != NULL)
            serv->total_requests++
        pthread_mutex_unlock(server_array_lock)

        // need to update count of requests since last healthcheck
        pthread_mutex_lock(health_lock)
        requests_since_healthcheck++
        if (requests_since_healthcheck >= R)
            pthread_cond_signal(wake_healthcheck)
        pthread_mutex_unlock(health_lock)

        if (serv == NULL)
            send 500 response to client
            close connection
            continue

        server_fd = connect_to_server(serv)
        if (server_fd == -1)
            send 500 response, close connection
            continue

        bridge_connection(client_fd, server_fd)

        // close the sockets
```

The healthcheck thread waits until receiving a signal from a worker thread to wake up or X seconds passes. It then wakes up the health worker threads to request their assigned servers and waits for all to update a temporary server array and

signal that they're finished. It then updates the main server_array, resets the count of requests since the last healthcheck and goes back to sleep.

```
healthcheck thread:
    // initialize as many healthcheck worker threads as there are servers
        // a copy of the server_array is made: temp_servers
        // each thread is given a pointer to a single object in this array
        // so no mutexes are necessary for this array
    // perform startup healthcheck, update server_array

    sem_post(health_init)    // signal to dispatcher that startup healthcheck has completed

    while true:
        pthread_cond_timedwait(wake_healthcheck, X seconds)  // wait for R requests or X seconds

        pthread_mutex_lock(worker_wake_lock)
        worker_wake_count = num_worker_threads         // count awake servers (see health worker thread)
        pthread_cond_broadcast(worker_wake)
        pthread_mutex_unlock(worker_wake_lock)

        pthread_barrier_wait(worker_barrier)    // wait for threads to complete healthchecks

        // update server array
        pthread_mutex_lock(server_array_lock)
        for (i = 0 --> server_count)
            servers[i] = temp_servers[i]

        pthread_mutex_unlock(server_array_lock)

        // update count of requests since last healthcheck
        pthread_mutex_lock(health_request_update_lock)
        requests_since_healthcheck = 0
        pthread_mutex_unlock(health_request_update_lock)

        pthread_mutex_unlock(server_heap_lock)
```

The healthcheck worker is given a pointer to a server_info object, which will be updated. This object is part of an array shared between the main health thread and all the workers. It sleeps until woken by the main healthhcheck thread and requests a healthcheck from its server. If it doesn't hear a response quickly enough or the response is bad, it marks the server as unresponsive. If the response is good, it updates the server info.

```
healthcheck worker:
    // intialize worker thread (grab all variables from health_worker_args)

    while true:
        pthread_mutex_lock(worker_wake_lock)
        while (worker_wake_count == 0)        // for spurious unblocks
        {
            pthread_cond_wait(worker_wake, worker_wake_lock)
        }
        worker_wake_count--
        pthread_mutex_unlock(worker_wake_lock)

        // request healthcheck from the assigned server
        perform_healtheck(server)

        pthread_barrier_wait(worker_barrier)
```

This function accepts a client/server socket pair and polls them for readiness and then calls a function to forward the data between them.

```
bridge_connection(client_fd, server_fd)
    while true:
        poll the sockets
        if (timeout && server never responded)
            send 500 error to client
            return
        if (ready to read from client)
            if (forward_data(client_fd, server_fd) == -1)    // -1 indicates server disconnect
                return
        if (read to read from server)
            if (forward_data(server_fd, client_fd) == -1)
                return
```

This function forwards data between 2 sockets

```
// return 0 for good execution or -1 if one of the sockets disconnected

int forward_data(in_fd, out_fd):
    while true:
        buf[BUF_SIZE]
        bytes = read(in_fd, buf, BUF_SIZE)
        if (bytes == -1 && in_fd disconnected)
            return -1
        else if (bytes == 0)
            break

        ret = write(out_fd, buf, bytes)
        if (ret == -1 && out_fd disconnected)
            return -1

    return 0
```

This function needs to search through the server_array to find the optimal server for routing requests to. It returns a pointer to the server_info struct of that server. It will perform a simple linear search because the server count is expected to be relatively small.

```
server_info* get_optimal_server(server_info* server_array, server_count:
    server_info* best_server = NULL

    for (i = 0 to server_count)
        if (server_compare(server_array[i], best_server))
            best_server = server_array[i]
```

This function compares 2 servers and returns true iff the first argument is more optimal than the second argument. A "more optimal" server has less total requests or a smaller ratio of failures/total_requests.

```
server_compare(server_info current_server, server_info best_server):
    if (best_server == NULL && !current_server->is_down)
        return true
    if (current_server->is_down)
        return false
    if (current_server->total_requests < best_server->total_requests)
        return true
    if (current_server->total_requests > best_server->total_requests)
        return false
    if (cur_ratio < best_ratio)
        return true
    return false
```

This function connects to the server, calls a function to send the healtheck, calls a function to check the message for good form, checks if its alive, updates the server data, and returns.

```
perform_healthcheck(server):
    server_fd = connect_to_server(server)

    char buf[HEALTH_RESPONSE_SIZE]
    if (get_healthcheck(server_fd, buf) == -1)
        server->is_down = true
        return
    if (!good_response(buf))
        server->is_down = true
        return
    // extract values from buf
    if (total requests < 0 or num failures < 0)
        server->is_down = true
        return

    server->is_down = false
    // update total requests and num failures in server
```

This function sends a healthheck to a server and puts the response in a buffer. HEALTH_REQUEST is a string defined at the top of my code. This pseudocode is greatly simplified.

```
get_healthcheck(serve_fd, buf):
    send(server_fd, HEALTH_REQUEST, strlen(HEALTH_REQUEST))

    while (not all bytes received)
        recv(server_fd, buf)
```

This function returns true if the health response is valid or false otherwise.

```
good_response(buf):
    if (protocol != "HTTP/1.1")
        return false
    if (status_code != 200)
        return false
    for every header:
        if header has bad form:
            return false
        if content-length not present:
            return false
    return true
```