Load Balancer Design Document

1. Introduction

Load Balancer is meant to act as a balancer of traffic to any amount of servers using N total connections. It is meant to health check these servers and use a heuristic to determine best server to send requests to.

1.2 Goals

- Run in Ubuntu 18.04
- Must be able to open any port above 8000.
- Must be able to handle any amount of servers
- Must be able to have N amount of concurrent connections set by administrator.
- Must be able to health check if R amount of requests occur.
- Must connect until client or server disconnect
- Must choose best server using lowest number of requests and lowest fails.
- Must take parameters in any order
- If all servers are down, respond to clients with code 500.

1.3 Statement of Scope

Below is the scope of the program, including the requirements and major inputs necessary to complete them.

1.3.1 Essential requirements:

- Must take in minimum 2 mandatory arguments: the load balancer port number and at least 1 server port number.
- Optional: N concurrent connection Count, and R request cap
- Must bridge client to best server and stay connected with no interference
- Must determine health of servers through a GET healthcheck
- Must estimate server load through internal book keeping
- Must be able to handle misaligned buffers, malformed headers, or incomplete health checks and other forms of network error.

1.3.2 Desirable requirements:

- No deadlock possibility and no race condition
- No possible way for the load balancer to fail
- Data security

1.3.4 Major inputs:

- *Argv[]*: Array of arguments consisting of the balancer port, server ports, and optional concurrent connection amount and request refresh threshold.
- *Argc*: amount of arguments

1.3.6 Major Constraints

- Buffer size must be limited to 16 kib
- Real time health is not knowable
- Must not use file pointers
- Health check is not synchronous, and threads could possible send data to an un-optimal server mid-check
- Must be in Ubuntu 18.04 exactly
- Must poll each server and timeout, which can cause major waits if all servers timeout.

2. Architecture and pseudocode with descriptions

Command Structure

```
typedef struct balancer {
        time structure for updater thread
        semaphore for updater thread;
        // total connections of servers
        int connections;
        // max amount of requests before refresh
        int max;
        // current number of requests waiting and its waitlist
        int waiting;
        request* waitlist;
        request* newest;
        // pointer to thread arry
        pthread_t* thread;
        // flexible array of server structs
        server* servers[];
}
```

The load balancer has a “two faced” command structure that act much like a differential in a car’s axel. From a central balancer object, the health check thread runs independently of the rest of the balancer and is the only one capable of updating crucial metadata. The other threads will only read that metadata, and make inferences based on it, when the updater is done updating the entire array. This allows both threads to work independently by only having 1 race condition in the entire load balancer and is impossible to dead lock.

Benefits:

- Less severe race conditions that endanger the data itself
- Easy to implement programmer side
- This allows connection threads to have current data without it being half written
- Fast access to data using a premade index and array system in the server structs.

Cons:

- Slow

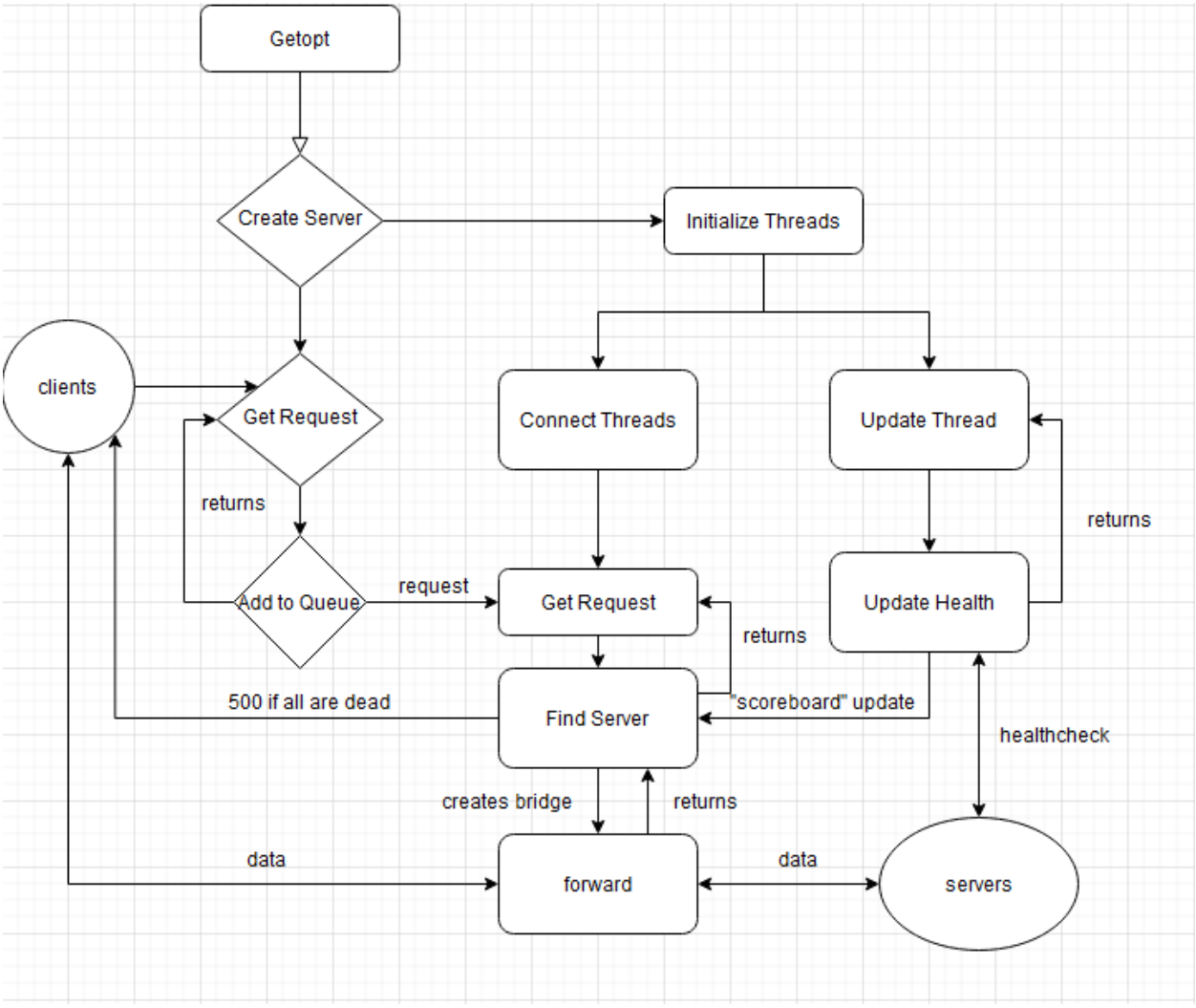
Server Structure

```
Server {
   // index to instantly find it in the balancer's array
   Const int id;
   // port number
   uint16_t port;
   // expected load between checks
   int requests;
   // actual reported health of the server
   double failure;
   // if its reachable
   bool status;
}
```

The server is a struct within the balancer object, where it stores the ID and port and all meta data necessary. The index is meant for general debugging and also some light operations. Failure is the failure rate of the server in percent. By using requests, the balancer can estimate what the new load is each time, not allowing the balancer to send all requests to one server until the next check. Similarly, a bool for status allows instant comparison and faster searching. The FD/Socket is not stored as those periodically close, but the port itself is saved for repeated use.

Overall Structure

The structure will rely on a two-faced system discussed before. By separating the connections and health checkers, it allows the removal of innumerable race conditions. The connections will not be able to interact with the health check in any meaningful way, but the updater will be able to update each server's metadata before the connection may fire findserver().



To avoid half written data, find server and update health is locked to the same mutex. This means only one thread is allowed to be in the scoreboard at any given time. This also solves the estimation race condition as only one thread can update a server's estimate at once. With this system, I estimate that there are also no dead locks as updater thread creation happens first, and find server cannot fire without a request or an OK from the updater.

TESTING

Testing was done with personal bash scripts to overload the queue and using a posted discord bash script from Aviv. The aviv discord scripts tested various scenarios of servers being up and down and whether the server could handle all forms of data sizes. Due to the nuance of checking if multiple connections were fielded, that test was done manually with printf statements showing every desired thread was created and handling requests.