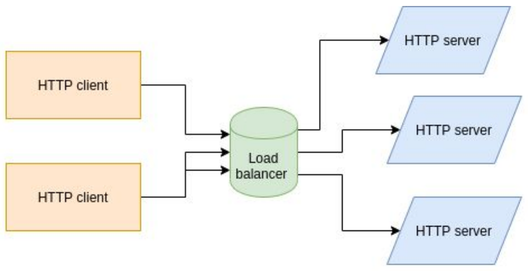
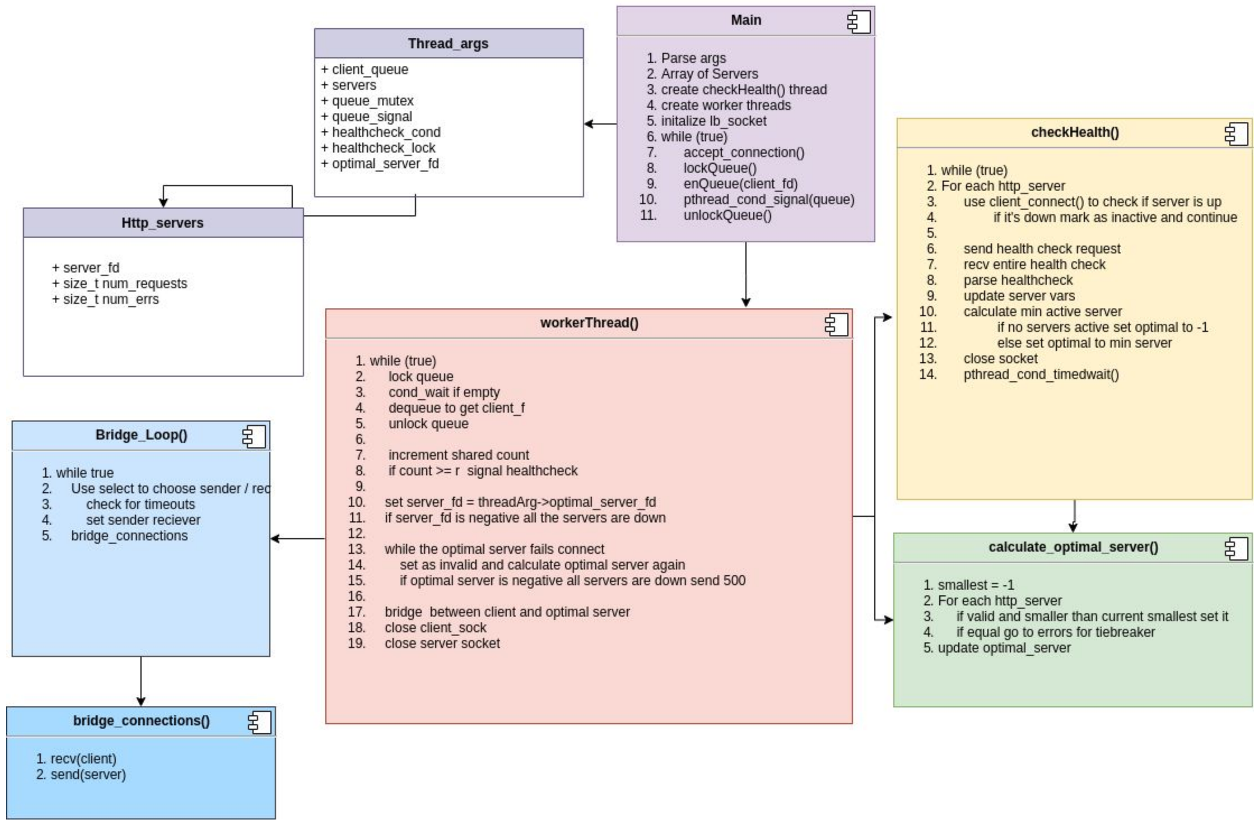
Introduction

a. An HTTP server load balancer that balances requests every X seconds or R requests, handles up to N connections at the same time





Thread_args

```
+ client_queue
+ servers
+ queue_mutex
+ queue_signal
+ healthcheck_cond
+ healthcheck_lock
+ optimal_server_fd
```

Http_servers

```
+ server_fd
+ size_t num_requests
+ size_t num_errs
```

Main

```
1. Parse args
2. Array of Servers
3. create checkHealth() thread
4. create worker threads
5. initalize lb_socket
6. while (true)
7.      accept_connection()
8.      lockQueue()
9.      enQueue(client_fd)
10.     pthread_cond_signal(queue)
11.     unlockQueue()
```

checkHealth()

```
1. while (true)
2. For each http_server
3.     use client_connect() to check if server is up
4.         if it's down mark as inactive and continue
5.
6.     send health check request
7.     recv entire health check
8.     parse healthcheck
9.     update server vars
10.    calculate min active server
11.        if no servers active set optimal to -1
12.        else set optimal to min server
13.    close socket
14.    pthread_cond_timedwait()
```

workerThread()

```
1. while (true)
2.     lock queue
3.     cond_wait if empty
4.     dequeue to get client_f
5.     unlock queue
6.
7.     increment shared count
8.     if count >= r  signal healthcheck
9.
10.    set server_fd = threadArg->optimal_server_fd
11.    if server_fd is negative all the servers are down
12.
13.    while the optimal server fails connect
14.        set as invalid and calculate optimal server again
15.        if optimal server is negative all servers are down send 500
16.
17.    bridge  between client and optimal server
18.    close client_sock
19.    close server socket
```

calculate_optimal_server()

```
1. smallest = -1
2. For each http_server
3.     if valid and smaller than current smallest set it
4.     if equal go to errors for tiebreaker
5. update optimal_server
```

Bridge_Loop()

```
1. while true
2.     Use select to choose sender / rec
3.         check for timeouts
4.         set sender reciever
5.     bridge_connections
```

bridge_connections()

```
1. recv(client)
2. send(server)
```

Data Decisions

1. For holding client connections waiting for a worker thread I use a queue
   a. The Queue implementation is the best here because it allows for a theoretical infinite amount of backlogged connections
2. For holding server ports I used an array of structs
   a. By creating an array of server structs I can easily pass all server information
      i. Server_struct :

            1. File descriptor
            2. Load
            3. Errors
            4. Valid
    b. The array has less overhead than the linked list implementation and since all available http servers are defined at the beginning of the program I saw no need for a resizable structure. By using a boolean I mark servers valid or invalid.
3. One threadArgs struct to hold information for all functions to share
    a. Array of servers
    b. Locks | signals
    c. Load balancer port
    d. Optimal server port and Index
    e. R for when to signal a healthcheck

Design Decisions

1) If the optimal server throws an error during connect_client() I have the optimal server marked invalid and recall the calculate_optimal function. We only send a 500 response if all servers have failed between health check requests.
2) Calculate_optimal function loops through the servers and finds the min server and sets it in the shared memory space. If all the servers are marked invalid the shared memory optimal server gets set to negative 1
3) I only use one mutex lock at the beginning of each worker thread to dequeue a client connection to work on. Since this is at the beginning of each thread daemon it causes minimal interruptions in service time to the client.
    a) *Note I do use another mutex in the end of healthcheck but it is a symbolic mutex so that I can use pthread_cond_timedwait to have a signal and a timer for starting health checks
        i) Since the healthcheck is a single thread there is no contention for the lock so is it even a real lock?
    b) Since the only data hazards between my worker threads is counting the number of client connections
        i) I can solve this by moving the incrementor to the main thread that accepts connections, now that it’s in a single threaded function I can remove this hazard
4) The load balancer is a front end service so I care more about speed and service to the client rather then calculating the *exact* optimal server for each request.
    a) I call it best guess service, who cares if the optimal server is outdated by a second, and since we take care of client connections that fail until we’ve checked all servers we know this best guess service won’t affect the amount of 500 responses we send.

5) In calculating the health_check data although I passed all the tests the write up question made me add a timeout to my health check so if we connect to a server but it fails to send a health check back we can mark it as problematic in 5s.

Testing

1) For testing I used a combination of bash scripting, and python scripting, and artisanal hand testing
    a) Bash scripting : takes args and performs different actions
        i) Creates 8 files using base64 /dev/urandom | head
        ii) Starts an load balancer
        iii) Starts N httpservers
        iv) Preforms 8 get requests N times
        v) Diffs the outputs to the created files
    b) Python scripting : worked on learning how the subprocess library works to create tests
        i) Used http request libraries to send a request in N chunks
    c) Hand testing
        i) I set the http timeout to a long time and used print statements to see the data for each httpserver struct and how my server was calculating the optimal server