Design Document: Load Balancing v1.0

Robert Hu

CruzID: ryhu

1.0 Introduction

1.1 Goals and objectives

The overall goal of this assignment is to program a functioning load balancer that is able to connect to multiple httpservers that follow the spec of the last assignment. The program must keep track of various variables and work to find the best way to sort incoming connections.

1.2 Description of assignment

The load balancer will act as an intermediary or middleman between the various servers and whichever client connects. It will act as a client for each server and as a server waiting for a connection from any client. This assignment will make use of methods used in the last assignment and adopt them to this assignment with some minor changes. It will use worker threads to handle parallel operation that receives requests and sends them to an appropriate server. The load balancer will perform healthchecks every 5 seconds or whenever a certain number of requests have been passed through. The program will store the healthcheck return information internally so the worker threads know where to send their requests.

1.2.1 Parallel operations

The default number of parallel operations that the load balancer can process at one time is 4. The program accepts a command line option to change the value to another number (at least 1). This number will determine the number of worker threads to create.

1.2.2 Request Count

The default number for the max request count is 5. The program accepts a command line option to change the value of the request count to another number (at least 1). When the program accepts a connection, it will increment an internal counter to keep track of how many requests have been sent. If the internal counter reaches the max request count, then the dispatcher will signal the healthcheck thread.

1.2.3 Timeout

Unlike the previous two functionalities, this one is a personal design choice that is set by myself. The other two are determined by an option with a default value.

For the timeout of the select() call that waits for a socket to read from as well as the timer for the healthcheck thread's timedwait() call is 3 seconds.

The reason why I chose to use 3 seconds as the timer is to limit the total number of healthchecks sent to each server as well as to give each server file descriptor and client file descriptor enough time to respond to each request. Using a timer that is too short would overflow servers with healthcheck requests, constantly keeping them locked, as well as preventing message passing between the client and the server. Using a timer that is too long would cause the load balancer to be unable to update server statuses for too long and is likely to cause imbalances in server load amongst the active servers.

2.0 Data Design

2.1 Global data structures

There are three structs created to handle all connections and any key information needed to handle requests.

2.2 Struct 1: server()

The server struct will hold: an int to hold the port that is associated with it, a bool to indicate whether or not it is alive, two ints to hold the total and failure requests received by healthchecks, a pthread_mutex_t lock to limit access to the struct during manipulation, and lastly an int to hold the initial server file descriptor returned by a client_connect call on the port.

2.2.1 Purpose and Reasoning behind server()

The purpose of the server struct is to contain the relevant information to handling server conditions. The variables allow the load balancer to keep track of request counts per server as well as their current status.

This struct was designed and implemented simply for holding relevant information just like the httpObject struct that was implemented in the last two assignments. The locks allow for only one thread to access them at any one time so as to prevent multiple threads from interfering with one another.

2.3 Struct 2: healthChecker()

The healthChecker struct holds 4 variables: a uint8_t buffer to hold the message acquired by the recv() call on the file descriptor connected to by a client_connect call on the appropriate server port, a pthread_mutex_t lock and pthread_cond_t condition_var to allow for signaling and timed waits, and a pthread_t thread id.

2.3.1 Purpose and Reasoning behind healthChecker()

The healthChecker struct is passed into the healthcheck probe thread to allow for pthread_cond_signal() pthread_cond_timedwait() calls to determine when a new healthcheck need to be called and processed as well as a buffer to hold the response to the healthcheck that will be parsed. If the returned status code is the correct 200, then the internal variables for each server are updated so that the next call to find the best server is using the most recent info.

The main reason behind this struct is for processing healthchecks whenever the load balancer determines that it is appropriate to do a healthcheck on the servers. Initially, I was not going to implement this struct, however, after testing my code a few times, I realized that I wouldn't be able to ensure that the healthcheck would not be interfered with by other requests passing through the load balancer.

2.4 Struct 3: parallel()

The parallel struct holds 5 variables: an int for the client socket received from an accept() call, an int id to identify which worker thread is handling the request, a pthread_mutex_t lock and pthread_cond_t condition_var to allow for locking, and a pthread_t thread id.

2.4.1 Purpose and Reasoning behind parallel()

The parallel struct is passed into the worker parallel operation threads to keep track of which client socket is the one connected to the client that is send the message that is being sent to the best server. In the main call, after accept() is called and returns, the client socket file descriptor is given to the first waiting worker thread.

The main reason behind this struct is precisely the client socket file descriptor to pass around. Originally, functionality to accommodate for locking the worker threads was intended to be implemented as I did in the previous assignment however, I ended up not using them.

3.0 Architectural and component-level design

3.1 System Structure

The overall structure of the program revolves around creating a thread to handle healthcheck probes as well as as many threads as are stated in the parallel thread option to handle the parallel operations. When the dispatcher accepts a connection, it will test the validity of the accept() file descriptor and check if it is necessary to perform a healthcheck. It then gives the file descriptor to a worker and waits for another connection. The worker handles the interaction between the accept file descriptor and the selected best server file descriptor. After the interaction, it closes and waits for the dispatcher to notify it of more work.

The worker threads use a function called handle_connection and the probe thread uses a function called probeHealthcheck. The worker threads are referred to as parallelOp threads in my implementation.

4.0 Starter Code description and implementation

4.1 Starter Code usage

From the starter code that is provided to us by the TAs, we have the basic functionality required to connect to server ports and well as creating a listening port to wait for incoming connections from clients. It shows how the load balancer will behave and lists some basic functionality that will allow communication between the servers and the client.

4.2 client_connect()

Client_connect() takes in a port number and establishes connection as a client to an httpserver. It returns a valid file descriptor for that server or it returns -1 if it fails on any step.
I did not modify this method in my implementation of the load balancer.

4.3 server_listen()

Server_listen() takes in a port number and creates a socket to listen in on for clients that are sending in requests. Similar to the above function, it returns a valid file descriptor or -1 if it fails.
I did not modify this method in my implementation of the load balancer.

4.4 bridge_connections()

Bridge_connections() is the main function responsible for reading and writing between sockets for communication. The original implementation allowed it to send and recv up to 100 bytes.
In my implementation, the main changes were to change from 100 to my predefined BUFFER_SIZE variable as well as to remove some sleep and printf commands so it executes cleanly.

4.5 bridge_loop()

Bridge_loop() forwards all messages between both the client and the server sockets. In the original implementation, it printed a message if both sockets timed out and were idle. This would have resulted in an endless loop of waiting for more messages.
In my implementation, I changed the waiting timeout to 3 seconds for faster processing. In the event of a timeout, I had the function send a default 500 response to the client and return to the function that called bridge_loop();

5.0 Main function implementation

5.1 Main function responsibility

The main function parses the command line for the options and valid port numbers, initializes some global variables, performs some error checking on the arguments, and declares and initializes the worker/parallelOp threads and the healthcheck probe thread as well as the values that they hold. Then in a while loop, it accepts connection requests, check for probe signal requirements and then forwards the connection to a worker thread.

5.2 Main function algorithm pseudocode

```
INPUT: argc, argv
OUTPUT: none

1. Initialize request counter, max requests, and parallel op count
2. Check for enough command line arguments; exits if not enough
3. Iterate through command line args for -N and -R options and set relevant args
   if options present
4. Check validity of port numbers and create listen port; exits if any check fails
5. Iterate through remaining ports to create all server ports
6. Iterate through all servers, test is alive and initialize variables
7. Create and initialize parallel worker structs and the appropriate variables
    a. Create pthread for each parallel struct; exit on fail
8. Create and initialize healthcheck probe struct and the appropriate variables
    a. Create pthread for the healthcheck struct; exit on fail
9. WHILE (true)
    a. Accept incoming connection
    b. Increment internal counter and check if counter equals max requests
        i. Signal healthcheck if so
    c. Determine best server
        i. if fail, send fail response and close accepted connection
        ii. continue to next connection
    d. Iterate through parallel worker threads and signal first available thread
       to work on the accepted connection
```

Algorithm 1: int main(int argc, char** argv)

6.0 Worker pthread function implementation

6.1 Worker pthread function responsibility

.The handle_connection() function takes in a parallel struct as an argument. It will wait for a signal from the main to show that it has a task to work with and try to connect. If it is successful, it will call bridge_loop() on the appropriate sockets; otherwise it will send the default 500 response and close the accept() socket. After handling the message passing, it closes both sockets and resets the thread's client socket file descriptor to wait for another signal.

6.2 Worker pthread function algorithm pseudocode

```
INPUT: parallel struct
OUTPUT: none

    1. Set reference parallel struct
    2. WHILE (true)
        a. wait for a signal indicating that there is work to be done
        b. attempt to connect to best port
        c. if failed to connect
            i. send fail response and close accept() socket
        d. else
            i. call bridge_loop() on both sockets
            ii. when returned from previous called function, close both sockets
        e. reset thread's socket variable and prepare to wait again
```

Algorithm 2: void* handle_connection(void* thread)

7.0 Healthcheck pthread function implementation

7.1 Healthcheck pthread function responsibility

The probeHealthcheck() function takes in a healthchecker struct as an argument and iterates through each server. It first checks if they are alive and accepting connections. If it is, then it will send a default healthcheck request to the server, parse the response and store the total and failed request counts. After attempting to connect to each server, it tries to determine and set the best server to use, sets the timed wait timer and waits for a healthcheck signal from the main function. Lastly, it is responsible to reset the internal request counter.

7.2 Healthcheck pthread function algorithm pseudocode

```
INPUT: healthcheck struct
OUTPUT: none

    1. Set reference parallel struct and timeval and timespec structs
    2. WHILE (true)
        a. Iterate through servers
            i. Lock mutex
            ii. Try to connect; if fail, set server to dead and unlock mutex
            iii. Send default healthcheck request to connected socket
            iv. Reset buffer
            v. WHILE (1): read from connected socket until nothing is left to
               read
            vi. Parse message received from socket
            vii. Set server total and fail counts to appropriate numbers
            viii. Unlock mutex
        b. Determine best server
        c. Get current time
        d. Set wait time
        e. Set timedwait condition and wait for either a signal or for the wait time
           to pass
        f. Reset internal counter
```

Algorithm 3: void* probeHealthcheck(void* thread)

8.0 Best server function implementation

8.1 Best server function responsibility

The determineBest() function will iterate through all servers created initially and check which server has the lowest total request count and/or highest success count. It will lock access to each server as it processes the information in that struct and will return the port number of the best server to forward connections to.

8.2 Best server function algorithm pseudocode

```
INPUT: none
OUTPUT: best available port

    1. Initialize bestPort, total count, and fail count variables
    2. for all servers specified on command line
        a. lock mutex
        b. check if server is alive
            i. if not, unlock mutex and go to next server
        c. if server total is less than total count
            i. set bestPort, total count, and fail count to appropriate server
               variables
        d. if server total is equivalent to total count,
            i. check if server fail  is less than fail count
                1. set bestPort, total count, and fail count to appropriate
                   server variables
        e. unlock mutex
    3. return port number of the best server
```

Algorithm 4: int determineBest()

9.0 Testing

10.0 Old design thoughts

THINGS I NEED TO IMPLEMENT

default 500 responses in case of errors

while loops in bridge loop/connection to handle multiple recv/sends change timeout response

OLD THOUGHTS

process for healthcheck from load balancer to servers

1. struct for each port representing each server
2. inside struct:
    a. port num
    b. client fd
    c. socket fd
    d. total num
    e. fail num
    f. success ratio = 1 - (fail num / total num)
    g. bool alive set to false initially
3. create get healthcheck message manually
4. send message to each server after every R requests or every X seconds
    a. default R = 5
    b. X is design choice
5. receive response from server
6. sscanf for first %d to check status code
    a. if status code is error then set alive to false otherwise set to true
7. sscanf response for format "%d\n%d" and save %d's into fail num and total num
8. calculate fail ratio

create threads to handle each server
main thread chooses best server to handle client request based on least total num and success ratio to break ties
    if there's still a tie, just choose one of them
if server doesn't send response, send preset default 500 response instead

if connect fails, send default 500 response