Gail Dylan Salak
CSE 130

Load Balancer Design Document

Overview

Goal: This program builds upon the multithreaded HTTP server from assignment 2. It assumes there is a working httpserver program (which has been provided as a binary) that will be used to create multiple servers for which our program will balance the workload between. In this way, our load balancer will act as a client for the actual HTTP servers, while acting as a server for the client that is actually sending the requests.

Our load balancer will be able to check the functionality of the HTTP servers through periodic healthchecks. Through these healthchecks, we will be able to determine if a server is still running or "dead." We know if a server is dead if for some reason, we do not receive a response within a certain time. Choosing between the live servers, we will prioritize sending requests to the server with the least amount of entries. As we repeat this process of requesting healthchecks and sending requests, we will ensure that the requests are somewhat balanced between our servers.

Usage: ./loadbalancer [client_port#] [server_port#]

- Client port must come first
- Must be at least one server port
- **Flags:**
    - -N [# of parallel connections]
        - Much like the threads from asgn2, this flag denotes how many connections can be concurrently serviced.
        - Without -N flag, program defaults to 4 connections.
    - -R [# requests]
        - This flag denotes that after R requests, the load balancer should request a healthcheck from the servers.
        - This healthcheck can either be processed every R requests, or after X seconds (which we will define later in the design).

Program Design

High-Level Description: Building off of the given start code, I will take advantage of the different functions to create the client/server connections and bridge these together through the load balancer. The main function will begin to parse the arguments through the getopt() function to find all the flags and given ports. The server ports will be placed in an array which we will later loop through. We will then create the connections needed for our loadbalancer and create the threads necessary by -N.

These threads will have a mutex lock on a requestCounter variable that will be shared between all threads. They will all increment the counter once they send a response from a server back to the client. A dedicated healthcheck thread will sleep on a timed wait condition to be signaled when either the requestCounter reaches the numRequests given by -R or a certain time elapses denoted by X. The healthcheck thread takes care of comparing the healthchecks of all the servers, through use of sscanf(), and then changing the target server to the least busy server. If a server doesn't respond to the healthcheck within X seconds, the load balancer will move on to the next server.

Global Variables and Definitions

#define X 2

- X is the number of seconds before a timeout. So, we wait X seconds for a response from a server and we wait X seconds between healthchecks. I chose 2 as of right now, because it is fast.

#define INTERNAL "HTTP/1.1 500 Internal Server Error\r\nContent-Length: 0\r\n\r\n"

- Internal error 500 string that can be sent easily to the client whenever servers are down or unresponsive

int requestCount = 0;

- global number of requests which is incremented when a worker thread has finished handling a request
- initialized to 0 since the program starts with 0 requests

We will create an array of servers through which we will be able to loop through for healthcheck probes and deciding to which server to send requests.

struct server {

- size_t port
    - # of server in array
- int connfd
    - the server port on which the load balancer works as a client
- size_t totalRequests
    - variable used to store and increment requests even between healthcheck probes
- size_t totalErrors
    - error count per server set at each healthcheck

- bool alive
    - flag that signals whether a server is up or down

}

We will create an array of worker threads that each take care of one request and bridging it to an appropriate server.

struct worker {

- ssize_t ID
    - # of thread
- int acceptfd
    - Client socket for thread
- size_t optServer
    - index of the calculated optimal server
- pthread_t workerID
    - WorkerID for creation of thread
- pthread_cond_t condition
    - Condition variable for thread to wake up when signaled by dispatcher
- pthread_mutex_t* lock
    - Lock on thread condition variable
- pthread_cond_t* dispatchCond
    - Condition variable to be signaled when the thread is done with a request
- pthread_mutex_t* requestLock
    - Lock on incrementing the requestCount variable and checking it versus R_value
- pthread_cond_t* hcCond
    - Condition variable within requestLock to be signaled when requestCount = R_value
- size_t portCounter
    - stores amount of server ports in server array
- struct server* servers[50]
    - So the worker threads can have access to the server ports
    - 50 is just a large number to initialize the array

}

We create a healthchecker struct with all the information needed for the hcProbe() function.

struct healthchecker {

- pthread_t hcID;
    - thread ID of hcThread
- pthread_mutex_t* hcLock;
    - lock for healthcheck to be waiting in
- pthread_cond_t* hcCond;

    - healthcheck has a timed wait on this conditional variable
- pthread_mutex_t* serverLock;
    - server lock on the server totalRequests and totalErrors so there is mutual exclusion with the dispatcher
- pthread_cond_t* optCond;
    - condition variable to signal the dispatcher when the healthcheck is done
- size_t portCounter;
    - stores amount of server ports in server array
- struct server* servers[50];
    - array of server structs and their information
    - initialized to 50 as a large improbable number

};

Starter Code: We were given starter code with 4 important functions to build from. I kept client_connect() and server_listen() exactly the same because they create the connections perfectly well.

In bridge_connections(), I increased the buffer size to 4096 so we could read in 4KiB at a time instead of the original amount of 100 bytes.

In bridge_loop(), under case 0 of the switch statement, I added a dprintf() statement that sends a 500 error to the client because this statement is reached when a server is not responding within X seconds.

main(): The main function begins with using getopt() to parse through the command line arguments. After getopt() parses through each argument, we should have one client port, at least one server port, and optional thread counts and request counts. If we don't all the mandatory arguments, we exit failure. If no optional flags are given through -N or -R, we default to 4 threads or 5 requests respectively. We had stored the server ports into an unbounded array, so after getopt() finishes, we store this array into a bounded array called serverPorts[] and free the previous portArray.

We then initialize an array of server structs which we conveniently call servers[]. In this array, each server is initialized with their totalRequest and totalErrors as 0. Their port is the corresponding port from serverPorts[]. Lastly, in this loop of initializing values, we try to connect to each server to determine if they are alive or dead. Depending on the return of client_connect(), we set the server's alive boolean to true or false. We then remember, to close the connfd.

Similarly, we move on to initialize an array of worker threads, which follows the same format as the worker thread struct array from assignment 2, except the threads have some additional information. These new variables include an acceptfd and optServer, both initialized to INT_MIN as they will be changed later once the thread is called. We pass the address of the requestLock and hcCond as the worker threads will use these. Importantly, we give each thread

access to the struct array servers[] which we previously made. The worker threads will later need this to connect to servers.

Lastly, we initialize one thread for the healthcheck probe which we call hcThread. We pass this a few different locks and condition variables. Much like the worker threads, we make sure to give the hcThread access to the servers[] array for comparison of servers.

Then, we enter a while loop that accepts connections. This acts in the same manner as my while loop from assignment 2, as it functions as a dispatcher thread that waits for a free thread and sends a request to that thread. We add some very important additions to this while loop. We have conditional statement to signal the hcThread which triggers if the requestCount is divisible by our R_value. This if statement is also locked by optLock because after we signal the hcThread, we wait for its response on optCond before proceeding.

We then calculate the optimal server through looping through the servers[] array and comparing servers which are alive based on totalRequests and totalErrors. From this loop, we can determine the server with the least load and consequently our target server. In another mutex, serverLock, we then increment this server's totalRequests as we are about to send it a request. We then pass the acceptfd and the optServer to the target threads and signal it to run.

handleRequest(): Each worker thread runs through this function just like my implementation in assignment 2. Instead of processing an HTTP message, we are sending the request to the optimal server and sending the response back to the client. We first try to client_connect() the optimal server's port. If this does not work, we mark the server as dead and send the client a 500 response. If it does work, we then bridge_loop() the acceptfd and the connfd returned from client_connect(). We then close the file descriptors and set accept_fd to INT_MIN for the next iteration.

Next, we have a mutex, requestLock, within which we increment the requestCount since we have just finished handling a request. Lastly, we signal the dispatcher that we are done.

hcProbe(): The healthcheck thread is the only thread that executes this function. It will stay in this function as it will continue to loop forever. This function mainly consists of a while loop. Before the loop, we initialize a few structs, timespec and timeval, which are needed for our method of timing between healthchecks. At the beginning of the loop, we enter a mutex, hcLock, in which we first memset our timespec variable, timeout, to 0 in order to ensure that it will be different from the last iteration. Then we gettimeofday() into our timespec variable, now. Using this information, we can set timeout's tv_sec member to now.tv_sec + X. This sets it so that X seconds from now, we will reach the timeout. Thus, we use this as the third argument in our timed wait on the hcCond so that the healthcheck will either run every X seconds or as it is signaled by the dispatcher. We then exit the mutex.

Next, we enter a for loop which essentially loops through the servers[] array, sending each server a healthcheck. In this loop, we first attempt to client-connect() to the specified server

port. If we cannot connect, we mark the server as dead and continue the loop. If we can connect, we proceed to send this server a healthcheck request.

Then, we enter another while loop which receives the response from the server. If recv() returns an error, we mark the server as dead and break. We recv() the response into a buffer which we simply sscanf() to parse the message for the returned entry and error counts. After getting these values, we enter a mutex, serverLock, in which we update the server's totalEntries and totalErrors. After exiting this mutex, we end the loop by memsetting the buffer back to 0. The loop exits once recv() returns a 0.

After exiting the recv() while loop, we close the server file descriptor. Once we have gone through this process for every server, we exit the for loop and signal the dispatcher that the healthcheck is done through optCond.