Natalie Petrosian
CSE 130 Principles of Computer Systems Design
Prof. Peter Alvaro

Assignment 3: Design Document

1.0 Introduction

In this assignment, we create a load balancer. It distributes connections over a set of servers while ignoring unresponsive ones. It utilizes concurrency in order to handle multiple requests at the same time. A health check is performed on these servers in order to decide which server will receive an incoming connection. Thus, the load balancer has two responsibilities: acts as a server to incoming connections *and* as a client to one of the running servers at any given time.

1.1 Goals and objectives

The goal for this assignment is to create a load balancer that distributes connections over a collection of servers while issuing health checks to ensure they are still operational and select the server with the lowest load. The load balancer works with an HTTP client, and handles incoming requests concurrently. The load balancer receives requests from clients and passes them on to an HTTP server. It periodically polls all the servers and issues a health check request to retrieve the total number of requests and failures in order to assess the best server.

1.2 Statement of scope

The HTTP client will always send requests that can be processed by the HTTP server. However, we also account for an unresponsive HTTP server. The first parameter to the load balancer is the port number responsible for receiving connections from the HTTP clients. This first parameter can be followed by the port numbers for the servers. A minimum of one client port number is required as well as a minimum of one (or more) server port numbers. The load balancer can also take an optional parameter -N, which defines the number N of parallel connections it can service simultaneously. The default value is N = 4. Another optional parameter is -R, which defines how many requests it processes between health checks. There are two criteria that causes the load balancer to issue a health check to the servers: after every R requests or after every X seconds. The determining factor in which of these criteria is executed is the one that occurs first. The load balancer should inspect the servers after every R = 5 requests by default. The number of X seconds was determined by the developer (1 second was determined optimal for this assignment). The following are some examples of how the load balancer is executed. In all of them, the port that receives requests from the client is 1234. The optional parameters (-N and -R) may appear in any order, however the client port number will always be the first specified port followed by the server port numbers. The program exits if any of the parameters are not positive integers.

```
./loadbalancer 1234 8080 8081 8082
```

```
./loadbalancer -N 5 8080 -R 3 8081

./loadbalancer 1234 -N 4 8081

./loadbalancer 1234 9000 -R 10 9001

./loadbalancer 1234 5678 -R 2 2468 -N 14
```

1.3 Major constraints

It was required that this program be written in C. Standard libraries for HTTP or FILE * calls were not permitted except for printing to the screen.

2.0 Connection Forwarding

The two functions that our load balancer is responsible for is forwarding requests received from clients to servers and receiving the responses from servers and forwarding them to the clients. The load balancer did not need to process the content received from the client and redirect the content to a server. The load balancer ensures that the server sends a response before a certain amount of time, or sends a 500 Internal Server Error response to the client. The amount of time the load balancer should wait until sending the error response was at the discretion of the developer. However, it could not be longer than five seconds. We can assume that if the server has started responding then the load balancer does not need to send the error response and can forward everything received from the server to the client. It sends all the data received from the client to the server and all the data received from the server to the client until one of them closes the connection.

2.1 Load Balancing

The load balancer is responsible for selecting the server with the lowest load. To make this possible, the program probes the servers with a health check at the start, and again every X amount of time or after every R requests received, whichever occurs first. The amount of time X was picked at the discretion of the developer (1 second). R can be specified as a parameter, but it also defaults to R = 5. It is important to note that if X is too short, then the load balancer would issue too many health checks. In addition, if X is too large, then it runs the risk of missing server failures. Due to the timed nature of the testing for this assignment, X could not exceed 5 seconds. The load balancer prioritizes the server that has received the lowest number of requests, with failures in mind. Should the lowest number of requests be the same in more than one server, the total number of failures in such servers is the tie-breaker. If the total number of failures are the same, then either server qualifies. If a server fails to respond to a health check from a client, it is marked as problematic and is excluded from balancing until a positive response is returned during another health check. A health check fails if the server doesn't respond, if the response code is not 200, or if the body does not have the errors\nentries format.

2.2 Health Check

It is assumed that the load balancer will be communicating with HTTP servers from Assignment 2. The health check is a special GET request to the HTTP server. The body of a health check request is of the form r\nf, where r is the total number of requests, and f is the total number of failures. Requests and failures are retrieved from the body using strtok(). The server with the best criteria will be selected.

3.0 Architectural and component-level design

The starter code provided the bulk of communication facilitated by the load balancer between the client and the server. The main() function was altered significantly to retrieve the parameters properly, provide a health check, select the best port, and continue operation. In addition, a function named threadsOfHealthCheckers() was created to launch parallel threads to all specified ports retrieving health check information. A function named findBestPort() was added to sift through the data returned by the health checker threads and select the port with the best criteria. A thread named server_get_health() was added to act as the threads that threadsOfHealthCheckers() launches.

3.2.Processing narrative (PSPEC) for component n

main(): This function establishes the proper socket connection for communication between the client and server. It contains an infinite while loop for continued operation. Before entering the while loop, the main() routine calls threadsOfHealthCheckers() and findBestPort() to select the best server port. It then enters the while loop, which maintains the communication between the client and the server on the selected port. However, once every X seconds (1 second) or R requests, it selects the current, best port again and refreshes the file handle. If no open ports are found, a 500 Internal Server Error is generated to the client.

threadsOfHealthCheckers(): This function launches N threads of health checkers at a time until all the specified port numbers have been polled. For example, if 10 port numbers were specified and N is set to 4, and one connection is taken by the client, it'll check three ports at a time for three times and the single remaining port at the end.

findBestPort(): This function sifts through the array of thread structures named 'args[]'. Each arg contains a port number, as well as the number of failures on that port, and the number of requests on that port. If a port did not respond during the polling, its port number will reflect an error condition. By scrutinizing all of the args, findBestPort() identifies the port with the lowest load, and returns the port number to the caller.

serverGetHealth(): This is a thread that threadsOfHealthCheckers() launches in parallel to poll the specified ports in parallel. It sends a health check GET request to a specified port and retrieves the response into the pertinent thread structure.

client_connect(): This function was provided in the starter code. It takes a port number and establishes a connection as a client. The server port number is provided as a parameter. It returns the client file descriptor.

server_listen(): This function was provided in the starter code. It takes a port number and creates a socket to listen on that port.

bridge_connections(): This function was provided in the starter code. It sends up to 100 bytes from fromfd to tofd.

bridge_loop(): This function was provided in the starter code. It forwards all messages between both sockets until the connection is interrupted. It toggles the connection between client and server depending on who is talking to whom. It utilizes a select() call for this purpose.

3.3 Design constraints for components

Standard libraries for HTTP and FILE * calls may not be used in any component.

4.1 Classes of tests

Two classes of tests should be performed on this program. Positive tests, which expect error-free executions of legal requests, and negative tests, which test the proper response of the server to erroneous conditions.

The following is a subset of tests that the program should handle appropriately.

GET a small file
GET a large file
PUT a small file
PUT a large file
Process a HEAD request
Handle concurrent GET/HEAD/PUT (difference resource)
Handle concurrent GET/HEAD (same resource)
Handle a 400 error
Handle a 404 error
Handle concurrent 400 and 404 requests
Forward a GET request to the least loaded server (2 servers have the same load)
Forward a PUT request to the least loaded server (2 servers have the same load)
Forward traffic to the least loaded server (each server has a different load)
Forward traffic to the least loaded server (many servers have the same load)
Handle concurrent connections from multiple clients
Detect a single server going down
Detect multiple servers going down
Detect if all servers are down initially (500 error)
Detect if all servers are non-responsive (500 error)

Detect a server going down and coming back up
Detect invalid health check responses (status code != 200)