CMPE476 - Distributed Systems

Project #2

Due June 4, 23:59

In this project, you **<u>are allowed to use</u>** ChatGPT or a similar large language model (LLM). However, make sure you cite the parts you acquired from LLM properly using inline comments. Also, provide the prompt you used to get that help from LLM. Note that you will not lose points depending on how much help you received from LLM, but I would like to see how good you are also in writing the prompts. Note that LLM systems often give incorrect results, so it is your responsibility to check it.

The programs should be developed using C language (using any version of gcc, but abiding 2011 standards) on Linux. Note that handling sockets in Windows is quite tricky, so I suggest that you use a virtual Linux machine if you are using Windows (even Mac OSX).

Note that the project document is also like the lecture notes on Load Balancers and Reverse Proxy. So, read the document carefully.

This project builds on top of Project #1. You may make use of the client and server code from the previous project (with necessary modifications), but you will need to implement the load balancer and the reverse proxy from scratch. The concepts for the load balancer and the reverse proxy are sometimes overlapping in functionality. However, they primarily focus on different aspects of network management and request handling

WHAT IS A LOAD BALANCER?

A **load balancer** is primarily responsible for distributing incoming network traffic across a number of backend servers or services to ensure that no single server becomes a bottleneck, thereby increasing reliability, availability, and performance. The key responsibilities of a load balancer are as follows:

1. **Traffic Distribution:** It evenly distributes client requests or network load to multiple servers based on various algorithms like round-robin, least connections, or IP hash. This helps prevent any single server from becoming overloaded.

2. **Health Checks:** Regularly checks the health of backend servers to ensure they are available to handle requests. If a server fails, the load balancer redirects traffic to the remaining operational servers.

3. **Fault Tolerance:** Increases the availability of applications by rerouting traffic from failed servers to other operational servers within the pool.

4. **Session Persistence:** Optionally maintains user session affinity (sticky sessions) by directing user requests from the same session to the same server, which is important for applications where session state is important.

WHAT IS A REVERSE PROXY?

A **reverse proxy**, on the other hand, sits in front of one or more servers and acts as an intermediary for requests from clients seeking resources from those servers. It can also serve as a load balancer but includes additional functionalities such as:

1. **Request Routing:** Directs incoming requests to the appropriate server based on content type, headers, or application data. This is particularly useful for directing traffic to different backend systems or services.

2. **Content Caching:** Stores copies of frequently accessed static content and serves this content directly to the client, reducing the load on backend servers and improving response times.

3. **SSL Termination:** Handles incoming SSL connections, decrypts requests, and forwards them to the backend servers in plain text. It can also re-encrypt the responses from the backend servers before sending them back to the client.

4. **Compression and Optimization:** Can compress outbound files and perform optimizations on the content to speed up load times and reduce bandwidth usage.

5. **Security Features:** Provides additional security by hiding the identities of the backend servers, implementing firewall policies, and performing DDoS protection measures.

KEY DIFFERENCES & OVERLAPPING FUNCTIONS

- **Function Overlap:** Both can perform load balancing tasks; however, a reverse proxy usually offers more advanced routing capabilities based on HTTP headers, cookies, or data within the requests.

- **Focus:** Load balancers are primarily focused on improving application availability and performance through basic network techniques, while reverse proxies provide deeper content management and security features at the application layer.

In summary, while both load balancers and reverse proxies help improve the efficiency and reliability of web services, reverse proxies offer more extensive control over traffic and content, making them suitable for more complex configurations and enhanced security setups.

WHAT ARE YOU GOING TO DO IN THIS PROJECT?

You will design a system composed of one load balancer, two reverse proxies, and three servers functioning under the control of each reverse proxy. Thus, there will be 1+2+2x3=9 applications running. Note that all of these applications will be created by a watchdog process that spawns these processes and tracks their proper processing. (See Note #1 below to understand what we mean by proper processing.) The watchdog, load balancer, reverse proxies, and all servers will print their logs on the same terminal where the watchdog is run. Therefore, they should all write the process name, such as *"[Reverse Proxy #1]: "*, *"[Server #4]:"*, etc., in the beginning of each line of output. You should also display explanatory comments on the screen (such as *"[Load balancer]: Started and ready"*, *"[Reverse Proxy #1]: Received a request from Client #107 and forwarding it to Server #5"*, *"[Server #5]: Received the value 4 from Client #107 and returning 16 as response"*, etc.). It is up to you to decide on the text of these messages, but it should be easy to track what is going on by looking at the log.

In addition to these processes, client processes will be run, each from a different terminal. Thus, there will be 9 system processes and N client processes running from 1+N different terminals on the screen.

As in the previous project, the server and client codes will be quite trivial for the sake of simplicity (although such a system would be built for a case where the server task is very heavy, and hence we try to distribute the load). The clients will be started from a terminal with a command line parameter that specifies the client ID. The client will read a floating point value from the user and send that value, together with the the client ID, to the server. However, it will be the load balancer who intercepts this request and forwards it to one of the reverse proxies, based on a hash of the client ID. (Odd numbered clients will always be sent to the first reverse proxy, and the even numbered clients to the second.) However, the reverse proxy should ensure that the value is non-negative (since the server is expected to take the square root) before relaying it to a server. If not, the reverse proxy will simply return -1 to the client via the load balancer. If the value sent by the client is legitimate (non-negative), the reverse proxy will pick one of the servers under its control randomly and send the request. The server simply calculates the square root of the value and returns it to the reverse proxy, which relays it to the client via the load balancer. The client should display the result from the server on the screen. Note that the server processes do not terminate after serving a client request, but each client asks a single question and terminates.

You will provide 5 separate source codes: **watchdog.c**, **load_balancer.c**, **reverse_proxy.c**, **server.c**, and **client.c**. Of course, there will also be a **Makefile** as before.

Note #1: By proper processing, we mean the watchdog tracks whether all of its spawned processes are up and running. If any process fails, the watchdog should print a log on the screen (starting with the text "[Watchdog]:"). If the watchdog receives the **SIGTSTP** signal, it should send **SIGTERM** signal to each of its spawned processes, starting at the servers, then the reverse proxies, and finally the load balancer. Each of these processes should print a log on the screen (in the form "[Server #2]: Received **SIGTERM** signal and exiting gracefully…"). Finally, the watchdog should also terminate. These requirements imply that you should handle the signals properly in the code. Research for signal handlers for this purpose.

Note #2: To send the SIGTERM signal to a process with process id PID, you can use the following command at the prompt:

```
kill -SIGTERM PID
```

where you can retrieve the process id PID for load_balancer using the command:

```
ps -ef | grep load_balancer | grep -v grep
```

The output of a sample run should be similar to below. Note that the user can enter input from any test window in the order they wish. Therefore, the requests arrive at the system in different orders.

Distributed System Window

```
$ ./watchdog
[Watchdog]: Started
[Watchdog]: Creating Load Balancer
[Load Balancer]: Started
[Load Balancer]: Creating Reverse Proxy #1
[Load Balancer]: Creating Reverse Proxy #2
[Reverse Proxy #1]: Started
[Reverse Proxy #2]: Started
[Reverse Proxy #1]: Creating 3 servers
[Server #1]: Started
[Reverse Proxy #2]: Creating 3 servers
[Server #2]: Started
[Server #3]: Started
[Server #4]: Started
[Server #5]: Started
[Server #6]: Started
[Load balancer]: Request from Client #107. Forwarding to Proxy #1
[Reverse Proxy #1]: Request from Client #107. Forwarding to Server #2
[Server #2]: Received the value 16 from Client #107. Returning 4.0.
[Load balancer]: Request from Client #112. Forwarding to Proxy #2
[Reverse Proxy #2]: Request from Client #112. Forwarding to Server #6
[Load balancer]: Request from Client #101. Forwarding to Proxy #1
[Reverse Proxy #2]: Illegal request from Client #101. Returning -1.
[Server #6]: Received the value 4 from Client #112. Returning 2.0.
[Watchdog]: Reverse Proxy has died. Re-creating.
[Watchdog]: Received SIGTSTP from user. Terminating all processes.
[Server #1]: Received SIGTERM from watchdog. Terminating.
[Server #2]: Received SIGTERM from watchdog. Terminating.
[Server #3]: Received SIGTERM from watchdog. Terminating.
[Server #4]: Received SIGTERM from watchdog. Terminating.
[Server #5]: Received SIGTERM from watchdog. Terminating.
[Server #6]: Received SIGTERM from watchdog. Terminating.
[Reverse Proxy #1]: Received SIGTERM from watchdog. Terminating.
[Reverse Proxy #2]: Received SIGTERM from watchdog. Terminating.
[Watchdog]: All processes terminated. Good bye.
```

Test Window #1

```
$ ./client 107
This is client #107
Enter a non-negative float: 16
    Result: 4.0
$ ./client 112
This is client #112
Enter a non-negative float: 4.0
    Result: 2.0
$ ps -ef | grep reverse_proxy | grep -v grep
user  4038  4009  0 15:03 pts/0   00:00:00 reverse_proxy
user  4236  4170  0 15:04 pts/1   00:00:00 reverse_proxy
$ kill -SIGKILL 4236
$ ps -ef | grep watchdog | grep -v grep
user  4009  4002  0 15:02 pts/0   00:00:00 watchdog
$ kill -SIGTSTP 4009
$ _
```

Test Window #2

```
$ ./client 101
This is client #101
Enter a non-negative float: -3
    Result: -1.0
$ _
```

Submission

You will submit the whole project as a single ZIP file in Moodle. Please stick with the naming convention. You will provide a Makefile for the whole project, which compiles each module when “make” is called. You should also enable compiling the watchdog, load_balancer, reverse_proxy, server, client separately using commands like “make watchdog”, “make load_server”, etc. The command “make clean” should remove all binary files.

You will also submit a report in the root folder, named report.pdf, which states the LLM prompts you used for each module and any problems you experienced. For example, if a specific LLM prompt caused an incorrect result, state it.