Build You Own Load Balancer

This challenge is to build your own application layer load balancer.

A load balancer sits in front of a group of servers and routes client requests across all of the servers that are capable of fulfilling those requests. The intention is to minimise response time and maximise utilisation whilst ensuring that no server is overloaded. If a server goes offline the load balance redirects the traffic to the remaining servers and when a new server is added it automatically starts sending requests to it.

Load balancers can work at different levels of the OSI seven-layer network model for example most cloud providers offer application load balancers (layer seven) and network load balancers (layer four). We're going to focus on a layer seven - application load balancer, which will route HTTP requests from clients to a pool of HTTP servers.

The Challenge - Building a Load Balancer

A load balancer performs the following functions:

- Distributes client requests/network load efficiently across multiple servers
- Ensures high availability and reliability by sending requests only to servers that are online
- Provides the flexibility to add or subtract servers as demand dictates

Therefore our goals for this project are to:

- Build a load balancer that can send traffic to two or more servers.
- Health check the servers.
- Handle a server going offline (failing a health check).
- Handle a server coming back online (passing a health check).

Step Zero

As usual this is where you select the programming language you're going to use for this challenge, set up your IDE and grab a coffee (or other beverage of your choice).

If you're not used to building systems that handle multiple concurrent events you might like to grab your beverage of choice and do some background reading on multi-threading, concurrency and asynchronous programming as it relates to your programming language of choice.

Step 1

In this step your goal is to create a basic server that can start-up, listen for incoming connections and then forward them to a single server.

The first sub-step then is to create a program (I'll call it '`lb`') that will start up and listen for connections on on a specified port (i.e. 80 for HTTP). I'd suggest you then log a message to standard out confirming an incoming connection, something like this:

```
./lb
Received request from 127.0.0.1
GET / HTTP/1.1
Host: localhost
User-Agent: curl/7.85.0
Accept: */*
```

For when I sent a test request to our load balancer like this:

```
curl http://localhost/
```

Next up we want to forward the request made to the load balancer to a back end server. This involves opening a connection to the back end server, making the same request to it that we received, then passing the result back to the client.

In order to handle multiple clients making requests you'll need to add some concurrency, either with your programming language's async framework or threads.

I decided to use a modified version of my code from the beginning of this step as my backend. It was changed to respond with a HTTP status of 200 and the text: 'Hello From Backend Server'.

Our test should look something like this:

```
./be
Received request from 127.0.0.1
```

```
GET / HTTP/1.1
Host: localhost
User-Agent: curl/7.85.0
Accept: */*

Replied with a hello message
```

```
./lb
Received request from 127.0.0.1
GET / HTTP/1.1
Host: localhost
User-Agent: curl/7.85.0
Accept: */*

Response from server: HTTP/1.1 200 OK

Hello From Backend Server
```

```
curl http://localhost/ --output -
Hello From Backend Server
```

Step 2

In this step your goal is to distribute the incoming requests between two servers using a basic scheduling algorithm - round robin.

Round robin is the simplest form of static load balancing. In a nutshell it works by sending each new request to the next server in the list of servers. When we've sent a request to every server, we start back at the beginning of the list.

So if we have three servers and six requests they get routed as follows:

Server	Requests
A	1, 4
B	2, 5
C	3, 6

You can read more about load balancing algorithms on the Coding Challenges website.

So to do this we'll need to do several things:

1. Extend our load balancer to allow for multiple backend severs.
2. Then route the request to the servers based on the round robin scheduling algorithm.

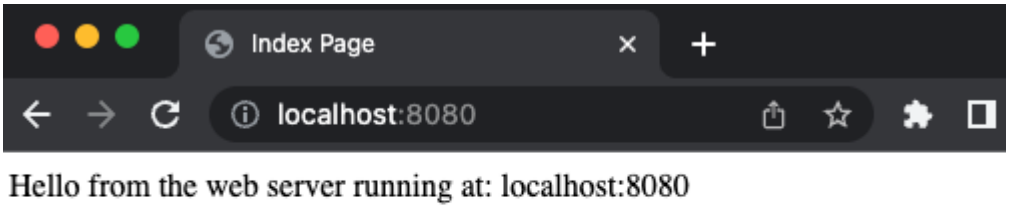
At this stage it's probably easier to use an off-the-shelf web server as our back end server. I'm going to use Python's built in web server in the following examples. On the Coding Challenges website I've documented how to run a simple web server with Python.

If you're happy to use Python and it's built in server please grab the zip containing the simple content we'll use from my dropbox here.

Unzip the content to your working directory and then run the Python HTTP Server as so:

```
% python -m http.server 8080 --directory server8080
Serving HTTP on :: port 8080 (http://[::]:8080/) ...
```

Then you can test it in your browser, which will give you this exciting page:



Simple HTML page returned from out backend test server

As we're going to balance requests between two servers you'll want to start a second server in another terminal. I'd suggest running that one on port 8081 like this:

```
% python -m http.server 8081 --directory server8081
Serving HTTP on :: port 8081 (http://[::]:8081/) ...
```

Which you can test in your browser again or with curl:

```
% curl http://localhost:8081/
<!DOCTYPE html>
<html lang="en">
        <head>
                <meta charset="utf-8">
                <title>Index Page</title>
        </head>
        <body>
                Hello from the web server running on port 8081.
        </body>
</html>
```

With two backend servers running on port 8080 and 8081 you're in a position to test access through your load balancer.

Hit your development environment of choice and create a list of servers - you can hard code them for now or provide a command line option to allow you to specify the initial set. Then when you get a request forward it to the next server according to the round robin scheduling algorithm.

With the two back end servers started up, you should then be able to send your requests to `http://locahost/` and see that the responses alternate between the servers on port 8080 and 8081.

Congratulations you've built a basic load balancer! Now to make it more useful it should only send traffic to healthy servers.

Step 3

In this step your goal is to periodically health check the application servers that we're forwarding traffic to. If any server fails the health check then we will stop sending requests to it.

For this exercise we're going to use a HTTP GET request as the health check. If the status code returned is 200, the server is healthy. Any other response and the server is unhealthy and requests should no longer be sent to it.

Typically the health checks are sent periodically, I'd suggest you make this configurable via the command line so we can set a short period for testing - say 10 seconds. You will also need to be able to specify a health check URL.

So in summary the key tasks for this step:

1. Allow a health check period to be specified on the command line.
2. Every period make a GET request to the health check URL if the result is 200 carry on. Otherwise take the server out of the list of available servers to handle requests.
3. If the health check of a server starts passing again, add it back to the list of servers available to handle requests.

It would be a good idea to run the health check as a background task, concurrently to handling client requests.

When it comes to testing this I suggest you start up a third backend server:

```
% python -m http.server 8082 --directory server8082
Serving HTTP on :: port 8082 (http://[::]:8082/) ...
```

Then connect to your load balancer three to six times to verify that it is rotating through backend servers as expected. Once you've verified that kill one of the servers and verify the request are only routed to the remaining two, without you, the end user receiving any errors.

Once you've verified that, start the server back up, wait just a little longer than the health check duration and then check it is now back to serving content when requests are made through the load balancer.

As a final test, check your load balancer can handle multiple concurrent requests, I suggest using curl for this. First create a file containing the urls to check - for this they'll all be the same:

```
url = "http://localhost"
url = "http://localhost"
url = "http://localhost"
url = "http://localhost"
url = "http://localhost"
url = "http://localhost"
url = "http://localhost"
url = "http://localhost"
```

Then invoke curl to make concurrent requests:

```
% curl --parallel --parallel-immediate --parallel-max 3 --config
urls.txt
```

Tweak the maximum parallelisation to see how well your server copes!

If that all works, congratulations, you've built a basic HTTP load balancer!

Beyond Step 3 - Further Extensions You Could Build

Having gotten this far you've built a basic working load balancer. That's pretty awesome!

Here are some other areas you could explore if you wish to take the project further and dig deeper into what makes a load balancer useful and how it works:

1. Read up about HTTP keep-alive and how it is used to reuse back end connections until the timeout expires.
2. Add some Logging - think about the kinds of things that would be useful for a developer, i.e. which server did a client's request go to, how long did the backend server take to process the request and so on.
3. Build some automated tests that stand up the backend servers, a load balancer and a few clients. Check the load balancer can handle multiple clients at the same time.
4. If you opted for threads, try converting it to use an async framework - or vice versa.

Share Your Solutions!

If you think your solution is an example of the developers can learn from please share it, put it on GitHub, GitLab or elsewhere. Then let me know - ping me a message via Twitter or LinkedIn or just post about it there and tag me.

Get The Challenges By Email

If you would like to recieve the coding challenges by email, you can subscribe to the weekly newsletter on SubStack here:

Tags: intermediate algorithms networking threading async