Project 1: HTTP Proxy and Load Balancer

Introduction

In this project, you will implement a combination of an HTTP proxy and a load balancer. The proxy will serve as an intermediary between clients and backend servers. Clients will connect to the proxy, which will forward client requests to one of the backend servers. This project will help you understand key networking concepts like connection handling, fault tolerance, scalability, and the role of proxies in distributed systems.

In this project, **you must use `epoll()` [1] to support concurrent connections.** Threads or `select()` are **NOT** allowed at all. `epoll()` allows programs to multiplex input and output through a series of file descriptors. More detailed instructions on how to use `epoll()` can be found here.

Be prepared: this is a single-person project and your skills will be exercised. So start early and feel more than welcome to ask questions. However, please note that the TAs are not allowed to debug your code for you during their office hours. They cannot touch your keyboard, and they will only spend a maximum of 10 minutes reading your code. This helps them to assist more students effectively. Therefore, it is recommended that you visit their office hours with specific questions, instead of vague statements like "my code doesn't work." For debugging, logging or tools like Wireshark[2] can be helpful. There are many ways to do this; be creative.

Proxy Functions

Your proxy is required to provide the following functionalities:

- It receives requests from clients and forwards them to backend servers in a load-balancing way. Typically, load balancing can be done at connection or request levels. In this project, **you must implement the load balancing mechanism for each request.** For example, if you have 3 backend servers, all the requests sent by each client need to be almost evenly distributed to the 3 servers.

[1] https://man7.org/linux/man-pages/man7/epoll.7.html
[2] https://www.wireshark.org/

- It needs to detect connection lost or server crashes and automatically redirect requests from failed servers to available ones. Note that your proxy will run HTTP 1.1 instead of HTTP 1.0, so your proxy should implement **persistent connections**.

  It must also support handling multiple client connections through `epoll()` and request/response **pipelining**. Be aware that due to pipelining, several HTTP requests or responses from different clients may be concatenated. Your proxy should manage these situations carefully.

Code Organization

1. `proxy.py` – Implementing the HTTP proxy and load balancer logic

2. `servers.conf` – A JSON file that contains all the IP addresses and port numbers of the backend servers. This argument must be passed to `proxy.py` from the command line. It should not be hard-coded. This file should have the following format. You can change the IPs and port numbers to your backend servers.

```
{
    "backend_servers": [
        {
            "ip": "127.0.0.1",
            "port": 8001
        },
        {
            "ip": "127.0.0.1",
            "port": 8002
        },
        {
            "ip": "127.0.0.1",
            "port": 8003
        }
    ]
}
```

Implementation and Testing Tips

1. Web Server Setup

To test your implementation, you need multiple instances of a web server. You can run them on different hosts that have different IPs. Alternatively, you can configure multiple web server instances on different ports on the same host. The following shows the detailed instructions for Nginx server configuration. You can also use other web servers, such as Apache or Tomcat.

- **Install Nginx**

```
sudo apt update && sudo apt install nginx -y  # Debian/Ubuntu
# or
sudo yum install nginx -y  # CentOS/RHEL
```

- **Create a New Nginx Site Configuration**

  To run multiple Nginx instances on different ports (e.g., 8001, 8002, 8003), create new files in `/etc/nginx/sites-available/`.

```
sudo vim /etc/nginx/sites-available/site_8001
```

  Paste the following configuration:

```
server {
    listen 8001;
    server_name localhost;

    location / {
        root /var/www/html;
        index index.html;
    }
}
```

  Repeat for other ports (8002, 8003).

- **Enable the Sites**

  Once the files are created, you must **symbolically link** them to `/etc/nginx/sites-enabled/`:

```
sudo ln -s /etc/nginx/sites-available/site_8001 /etc/nginx/sites-
enabled/
```

  Repeat for the other files.

- **Test the Configuration**

  Check for syntax errors:

```
sudo nginx -t
```

If you see:

```
nginx: configuration file /etc/nginx/nginx.conf test is
successful
```

Then, the configuration is correct.

- **Restart Nginx**

  Now restart Nginx to apply the changes:

```
sudo systemctl restart nginx
```

2. Request Identifier

One way to implement load balancing for each request is to **insert a unique identifier** into the HTTP request headers. When the backend responds, it includes the same identifier, which allows the proxy to map it back to the correct client.

An example HTTP request header can look like below:

```
GET /index.html HTTP/1.1
X-Request-ID: 123e4567-e89b-12d3-a456-426614174000
Host: example.com
User-Agent: Python-Client
```

To make sure that this custom header field is preserved by the Nginx server so that it also appears in the response packets, you need to modify your server block to allow `X-Request-ID`:

```
server {
    listen 8001;
    server_name localhost;

    location / {
        proxy_set_header X-Request-ID $http_x_request_id;
    }
}
```

Then, restart the Nginx server to apply the changes.

3. Testing with Apache Benchmark

Assuming your proxy is running on **127.0.0.1** and your **proxy listens on port 9000**, you can send test requests using Apache Benchmark (ab):

```
ab -n 500 -c 10 -k http://127.0.0.1:9000/
```

Alternatively, you can also **test HTTP pipelining** by using `wrk` [3] instead of `ab`:

```
wrk -t10 -c1000 -d10s --latency http://127.0.0.1:9000/
```

`wrk` properly handles pipelined requests and responses, unlike `ab`.

Useful Links

Programming in Python

The 8th edition of the textbook (and what we've discussed in class) uses sockets in Python. A Python socket tutorial is http://docs.python.org/howto/sockets.html

It may also help by reading the system implementation of epoll at https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysv/linux/sys/epoll.h.html. You will better understand the epoll events.

REMINDERS

- Submit your code to Canvas in a zip file
- If your code does not **run**, you will not get credits.
- Document your code (by inserting comments in your code)
- DUE: 11:59 pm, Friday, Feb 14th.

[3] https://github.com/wg/wrk