Experiment 1: DDoS Attack

Web-App

In this experiment, I will be focusing on my personal website/portfolio to understand its resilience against a DDoS attack. By simulating a DDoS attack using Selenium WebDriver, I aim to stress test the server and evaluate its ability to handle a high volume of requests. This will provide valuable insights into the appropriate DDoS prevention mechanisms to use, considering the average number of users accessing the application.

Metrics

Two primary metrics will be used to assess the server's performance during the simulated DDoS attack:

Load average: The load average is a measure of the average number of processes waiting for CPU time over a given period. A high load average value indicates that the server is struggling to keep up with the demand, which may lead to slow response times and degraded performance. Monitoring the load average during the attack will help us determine the point at which the server becomes overwhelmed.

%CPU(s): The %CPU(s) metric indicates the percentage of CPU resources utilized by the server. A high %CPU(s) value means that the server's CPU is under heavy load, and the system may experience performance issues. By observing the %CPU(s) during the simulated DDoS attack, I can identify the server's capacity to handle increased load and determine the threshold at which the server's performance starts to degrade.

Setup

To perform this experiment, I set up the following components:

Selenium WebDriver: I used Python to simulate a DDoS attack by sending multiple requests to the personal website/portfolio. The Selenium script will run in a loop, sending requests with varying sleep durations to simulate different levels of traffic intensity.

To install the web driver, I followed these steps:

- Open the Chrome browser.
- Click on the three vertical dots located in the upper-right corner of the browser window.

In the drop-down menu, hover over "Help" and then click on "About Google Chrome."

- The "About Google Chrome" page will open, and it will display the current version of your Chrome browser. The browser will also automatically check for updates and install them if available.

Then, I went to "https://chromedriver.chromium.org/downloads" to download the driver that matches my version and OS, and placed the file in the same directory as ddos_simulation.py

Python script (ddos_simulation.py):

```python
from selenium.webdriver import Chrome
import time

driver = Chrome("chromedriver.exe")
sleep_durations = [1, 0.5, 0.25, 0.1, 0.05, 0.01]

for sleep_duration in sleep_durations:
    for i in range(20):
        url = 'http://54.160.115.159:3000/'
        driver.get(url)
        time.sleep(sleep_duration)
```

Server-side metrics monitoring: On the server, I ran a script to record the load average and %CPU(s) metrics at one-second intervals during the experiment. The collected data will be saved in a file (metrics.txt) for further analysis. Then, %CPU (s) was normalized to fit the load_average data.

Server-side script (record_metrics.sh):

```bash
#!/bin/bash
OUTPUT_FILE="/usr/games/testdir/metrics.txt"
echo "Load Average (1m), %CPU(s)" > $OUTPUT_FILE

while true; do
  LOAD_AVG=$(uptime | awk -F 'load average: ' '{print $2}')
  CPU_PERCENT=$(ps -e -o pcpu --sort=-pcpu | head -n 2 | tail -n 1)
  echo "$LOAD_AVG, $CPU_PERCENT" >> $OUTPUT_FILE
  sleep 1
done
```

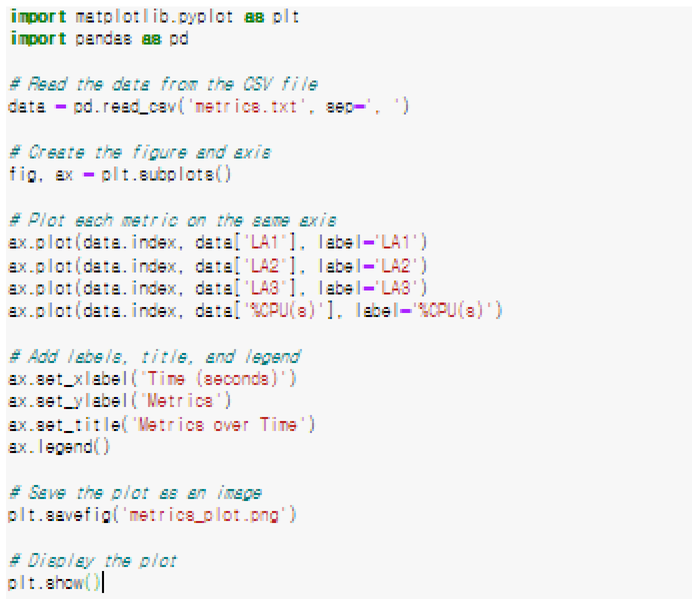
Execution:
With both the Selenium WebDriver script and the server-side script ready, I started the server-side script first to begin collecting metrics. Then, I ran the Selenium WebDriver script to initiate the simulated DDoS attack.

To run the server-side script I prepared:
chmod 777 record_metrics.sh

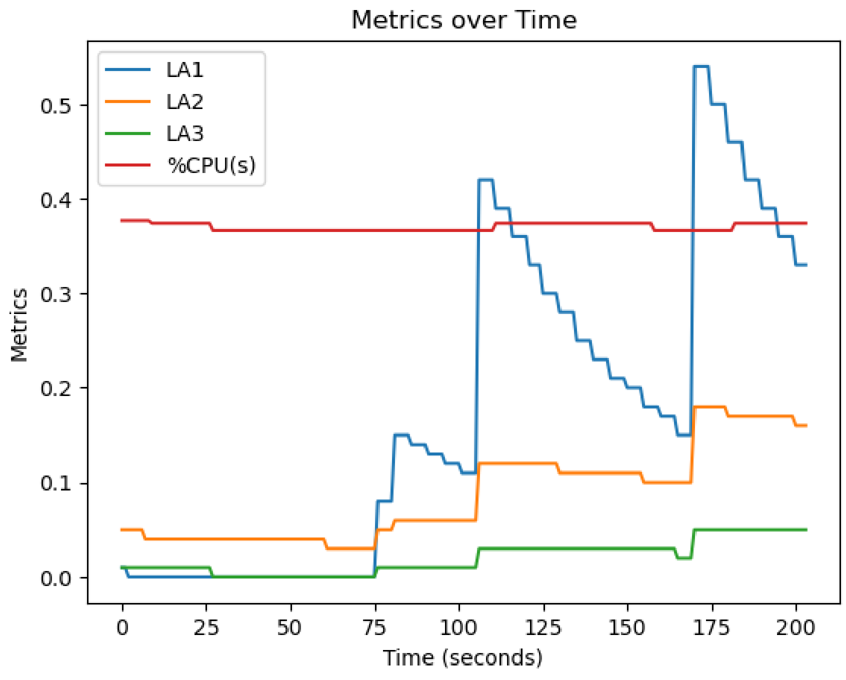
Then,
./record_metrics.sh

To run the Selenium WebDriver script, I ran:
python ddos_simulation.py

Results



```
import matplotlib.pyplot as plt
import pandas as pd

# Read the data from the CSV file
data = pd.read_csv('metrics.txt', sep=', ')

# Create the figure and axis
fig, ax = plt.subplots()

# Plot each metric on the same axis
ax.plot(data.index, data['LA1'], label='LA1')
ax.plot(data.index, data['LA2'], label='LA2')
ax.plot(data.index, data['LA3'], label='LA3')
ax.plot(data.index, data['%CPU(s)'], label='%CPU(s)')

# Add labels, title, and legend
ax.set_xlabel('Time (seconds)')
ax.set_ylabel('Metrics')
ax.set_title('Metrics over Time')
ax.legend()

# Save the plot as an image
plt.savefig('metrics_plot.png')

# Display the plot
plt.show()
```



After analyzing the data collected during the experiment, I could observe the following:

- The load average increases as the sleep duration between requests decrease, indicating a higher load on the server due to increased traffic intensity.

- The %CPU(s) utilization remains relatively low throughout the experiment, suggesting that the server's CPU is not the primary bottleneck in this setup.

- The highest load average reaches 0.54, which indicates that the server can handle a moderate amount of traffic without becoming significantly overwhelmed. However, as the traffic intensity increases, it is expected that the load average will rise, leading to degraded performance.

Considering the results obtained, it is evident that the server is capable of handling a moderate amount of traffic without significant performance issues. However, the experiment also highlights potential bottlenecks in the system, such as the server's inability to scale with increased traffic intensity. This implies that, under a real DDoS attack, the server might struggle to maintain performance.

To ensure better performance and resilience against DDoS attacks, it is advisable to upgrade the server instance or configuration. Upgrading to a server with more CPU resources and memory would enable it to handle higher traffic loads and maintain performance under stress. Additionally, implementing DDoS prevention mechanisms such as rate limiting, IP filtering, or using a Content Delivery Network (CDN) can further protect the server from potential attacks.

In conclusion, the results of this experiment indicate that the server can handle moderate traffic loads but would benefit from an upgraded instance or configuration to ensure better performance and resilience against DDoS attacks.

Experiment 2: Apache VS Nginx Comparison

Web-App

Continuing from Experiment 1, I am interested in further upgrading my creative project (personal website/portfolio) by adding a file-share function (fileshare.php), embedding a search engine (birthdaycard.html), and adding more pictures. Before moving on to implementation, benchmarking the files that already have the functions implemented would provide helpful insight into which web server to use.

Metrics

Requests per second

Requests per second (RPS) is a measure of how many requests a web server can handle within one second. This metric is essential for understanding the scalability and efficiency of a web server, as it directly reflects the server's ability to process incoming requests from users. A higher RPS value indicates that the server can handle a more significant number of requests simultaneously, leading to better performance and a more responsive website. In our benchmarking tests, I used RPS to evaluate the performance of Apache and Nginx when serving different types of files, such as HTML, PHP, and JPG files.

Time per request

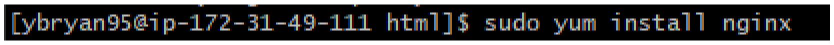
Time per request is the average time taken by the server to process a single request, measured in milliseconds (ms). This metric helps assess the latency and responsiveness of a web server, providing insight into how quickly users can access the website's content. A lower time per request signifies that the server is more efficient in delivering content to users, resulting in a faster and more seamless browsing experience. In our benchmarking tests, I compared the time per request for Apache and Nginx to determine which web server offers better performance when processing various file types.

By analyzing the benchmarking results using these two metrics, I can effectively evaluate the performance of Apache and Nginx in handling the added features to our personal website/portfolio. This information will help us make an informed decision on which web server to use for our project, ensuring optimal performance and an enhanced user experience.

Setup

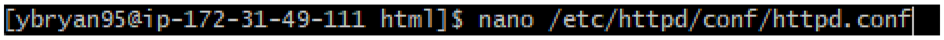
Apache was previously installed and set up from the Module 2 assignment. The instruction will follow guidelines on how to change proxy from Apache to Nginx.

First, install Nginx using the command,



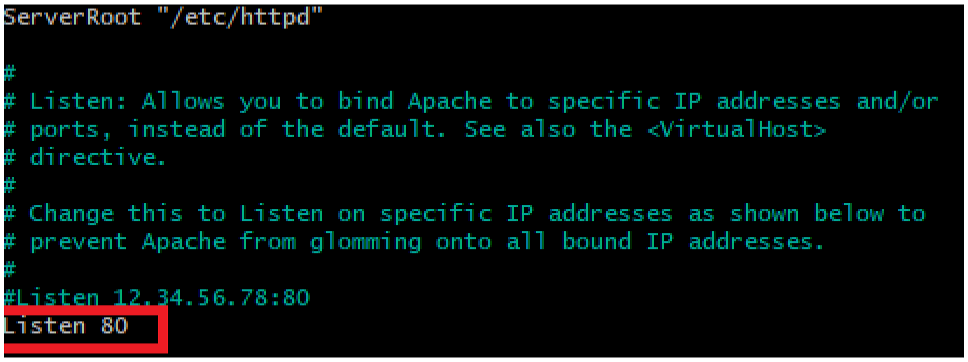
```
[ybryan95@ip-172-31-49-111 html]$ sudo yum install nginx
```

The next step can vary. For my project, I chose to replace Apache with Nginx by changing the port number used. In this case, find what security group you are using for Apache.



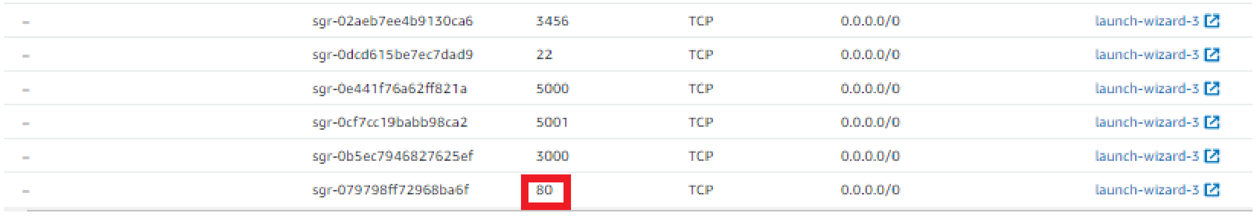
```
[ybryan95@ip-172-31-49-111 html]$ nano /etc/httpd/conf/httpd.conf
```

Above command will take you to a configuration page for Apache. Find your port number.



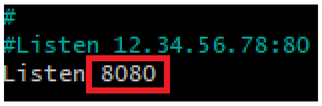
```
ServerRoot "/etc/httpd"

#
# Listen: Allows you to bind Apache to specific IP addresses and/or
# ports, instead of the default. See also the <VirtualHost>
# directive.
#
# Change this to Listen on specific IP addresses as shown below to
# prevent Apache from glomming onto all bound IP addresses.
#
#Listen 12.34.56.78:80
Listen 80
```

In this case, the port that was used is 80.
Then, confirm that information on your AWS EC2 Security Group



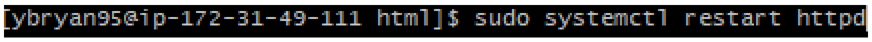
–	sgr-02aeb7ee4b9130ca6	3456	TCP	0.0.0.0/0	launch-wizard-3
–	sgr-0dcd615be7ec7dad9	22	TCP	0.0.0.0/0	launch-wizard-3
–	sgr-0e441f76a62ff821a	5000	TCP	0.0.0.0/0	launch-wizard-3
–	sgr-0cf7cc19babb98ca2	5001	TCP	0.0.0.0/0	launch-wizard-3
–	sgr-0b5ec7946827625ef	3000	TCP	0.0.0.0/0	launch-wizard-3
–	sgr-079798ff72968ba6f	80	TCP	0.0.0.0/0	launch-wizard-3

Modify the .conf file so that Apache listens to the port you are currently not using.



```
#
#Listen 12.34.56.78:80
Listen 8080
```

After modifying, exit the ‘nano’, and restart Apache.



```
[ybryan95@ip-172-31-49-111 html]$ sudo systemctl restart httpd
```

The 3 files I am testing are birthday.html, fileshare.php, and logo.jpg. I create a separate directory to copy these files from ‘/var/www/html’ (Apache virtual host configurated) to ‘/var/www/nginx’ (nginx virtual host configurated).

Using ‘sudo nano /etc/httpd/conf.d/apache-vhost.conf’, create the following content. This is the virtual host config for Apache. (In case separate folder for apache to be created, /var/www/apache can be used.)

```
<VirtualHost *:8080>
    DocumentRoot /var/www/apache
    <Directory /var/www/apache>
        Options Indexes FollowSymLinks
        AllowOverride None
        Require all granted
    </Directory>
</VirtualHost>
```

Then, perform ‘sudo nano /etc/nginx/conf.d/nginx-vhost.conf’. Insert the following content.

```
server {
    listen 80;
    server_name your_server_name;
    root /var/www/nginx;

    location / {
        try_files $uri $uri/ =404;
    }
}
```

Restart nginx and apache using ‘sudo systemctl restart nginx’ and ‘sudo systemctl restart httpd’.

Perform the following for files in apache-configurated folder.
‘ab -kc 1000 -n 10000 http://server_name:8080/filename’

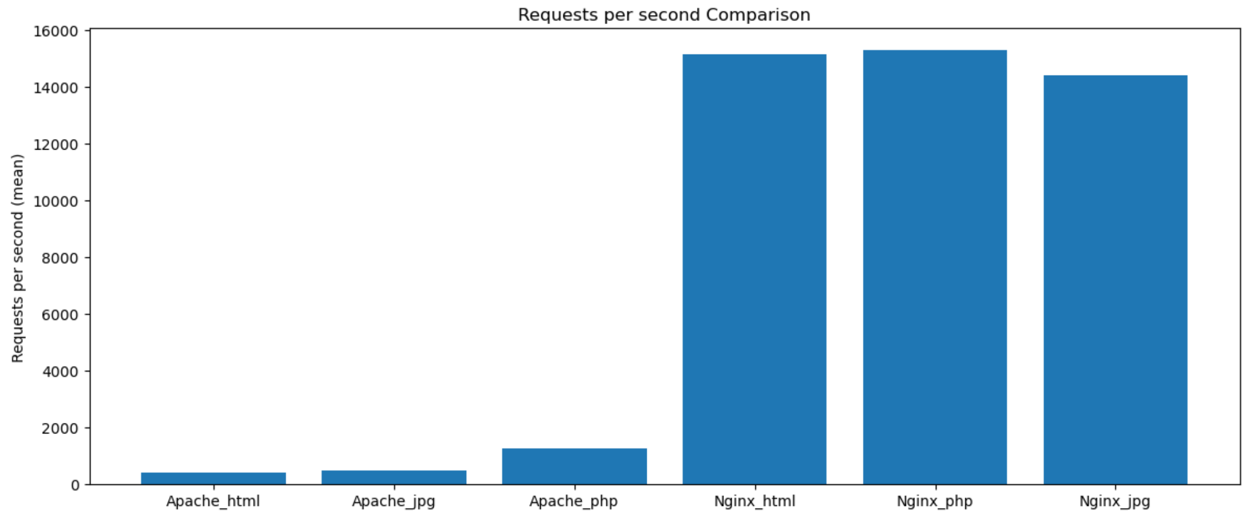
Perform the following for files in nginx-configurated folder.
‘ab -kc 1000 -n 10000 http://your_server_name:80/birthday.html’

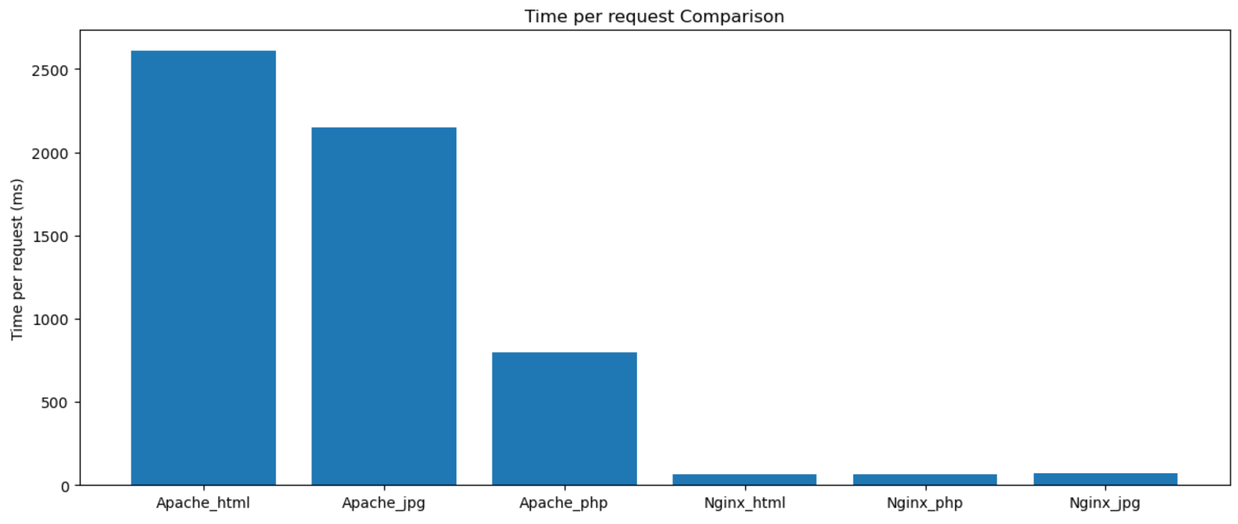
Results

From performance benchmarking on birthday.html, fileshare.php, and logo.jpg, I’ve obtained 6 results. All results are saved as ‘Apache_html’, ‘Apache.jpg’, ‘Apache_php’, ‘nginx_html’, ‘nginx_php’, and ‘nginx_jpg’ with jpg extension and uploaded to GitHub.

‘Apache_html’
Requests per second: 383.63 [#/sec] (mean)
Time per request: 2606.651 [ms] (mean)
‘Apache.jpg’
Requests per second: 465.08 [#/sec] (mean)
Time per request: 2150.147 [ms] (mean)

‘Apache_php’
Requests per second: 1250.66 [#/sec] (mean)
Time per request: 799.580 [ms] (mean)
‘Nginx_html’
Requests per second: 15129.81 [#/sec] (mean)
Time per request: 66.095 [ms] (mean)
‘Nginx_php’
Requests per second: 15296.81 [#/sec] (mean)
Time per request: 65.373 [ms] (mean)
‘Nginx_jpg’
Requests per second: 14413.52 [#/sec] (mean)
Time per request: 69.379 [ms] (mean)





The results obtained from the benchmarking tests on birthday.html, fileshare.php, and logo.jpg reveal significant differences in the performance of Apache and Nginx web servers. In all three cases, Nginx demonstrates substantially higher Requests per second (RPS) and lower Time per request values compared to Apache.

Given the performance differences between Apache and Nginx, it's essential to consider potential bottlenecks that may affect the system. For Apache, the bottlenecks might be related to its process-based architecture, which may struggle to handle numerous simultaneous connections, leading to increased latency and lower performance. Furthermore, Apache's modular design, which allows for various additional features and customization, can contribute to increased resource consumption and slower response times.

On the other hand, Nginx uses an event-driven architecture that can handle a large number of concurrent connections with minimal overhead, resulting in better performance and lower latency. However, it's essential to consider that Nginx may face bottlenecks when dealing with dynamic content or applications that require extensive server-side processing, as its primary focus is on serving static content efficiently.

Based on the benchmarking results and the analysis of potential bottlenecks, I recommend using Nginx as the web server for the upgraded personal website/portfolio. Nginx consistently demonstrates superior performance in handling the added features such as file-sharing, search engine embedding, and additional pictures. Its high RPS values and low Time per request values suggest that Nginx will provide a more responsive and efficient browsing experience for users compared to Apache.

However, it's crucial to monitor the performance of the Nginx server and watch for potential bottlenecks as the website grows and evolves, especially when implementing more dynamic content or server-side processing. This will ensure optimal performance and a consistently enhanced user experience.