CSCD58 Final Project:

Load Balancer

Pritish Panda (1005970914)

Yuto Omachi (1006005163)

Shannon Budiman (1006863770)

University of Toronto Scarborough

2 December 2023

Github Repository link:

- https://github.com/Lordpritish/LoadBalancer

Demo Video link:

- https://utoronto-my.sharepoint.com/:v:/g/personal/yuto_omachi_mail_utoronto_ca/EdwxQUEvCGVFv2u2s50PC-kBreWSLOT2v9FLd8eY2trhiQ

TABLE OF CONTENTS

Description and explanation of project

This project aims to implement a load balancer using Software-Defined Networking (SDN) principles, with a specific focus on the POX SDN controller and the Mininet network emulator. The rationale behind this project is to explore how SDN can enhance network performance through load balancing and gain practical experience in SDN application development. This project will include the ability to dynamically select and run different load balancing algorithms.

The project includes developing multiple load balancing algorithms such as Random Balancer, Round Robin, Weighted Round Robin, and Least Response Time. The Random Balancer algorithm selects servers randomly, offering simplicity and unpredictability. The Round Robin method distributes requests sequentially amongst servers, ensuring an even distribution of load. The Weighted Round Robin approach assigns weights to servers based on their capacity or other criteria, giving more requests to higher-capacity servers. The Least Response Time algorithm is more dynamic, selecting servers based on their current response times and workload, optimizing for the fastest possible response.

Dynamic algorithm selection is a key feature of this project, allowing network administrators to switch between different load balancing strategies based on current network conditions, traffic patterns, or specific application requirements. This adaptability is crucial for modern networks that face varying types of load and traffic demands.

To implement and test these concepts, the POX controller and Mininet emulator are utilized. On Mininet we run an s1 OpenFlow Switch with a remote controller that's run by Pox so that the switch can act like a Loadbacner. Where POX provides an open-source development platform for Python-based SDN control applications, such as the load balancer in this project. Mininet, on the other hand, creates a realistic virtual network, running real kernel, switch, and application code, for developing and testing SDN applications.

Contributions

Pritish was responsible for enhancing the load balancer with the dynamic algorithms. His work involved integrating multiple load balancing strategies, including Round Robin, Weighted Round Robin, and Least Response Time algorithms. This allowed for users to switch between different algorithms, which offered flexibility and adaptability to varying network conditions. These algorithms ensured efficient distribution of network traffic by taking server load and response times into account.

Yuto was responsible for developing the UI component that was crucial in visualizing the traffic managed by the load balancer, providing a concise summary of how the traffic was distributed across different servers. This allowed users to efficiently understand and analyze the performance of the load-balancing system, through intuitive graphics and charts. The UI provided a straightforward way for users to view the status of each server, understand the load balancing logic in action, and identify any potential issues in the network traffic distribution.

Shannon was responsible for setting up the basic load balancing mechanism based on a random selection algorithm, which ensured that each server had an equal chance of being selected, and provided a fair distribution of the incoming network traffic. Additionally, Shannon contributed to the development of testing scripts that simulated client requests to the load balancer, and also a server script.

Basic Idea and Flow

Load Balancer Operation:

1. Continuous Server Health Check:
    a. The load balancer employs a continuous health-check mechanism by pinging servers at regular intervals, typically every few seconds. This ensures that the load balancer is aware of the current status of each server in the pool.
2. Client Request Handling:
    a. Upon receiving a client request, the load balancer checks whether the client is not a server itself. This initial check ensures that server-to-server and dire client-server communication is avoided. All the requests should go to the load balancer only.
3. Server Selection and Request Forwarding:
    a. Using a predefined load balancing algorithm, the load balancer selects an appropriate server from the pool. This decision is based on load balancing algorithm,
    b. The selected server is then assigned to handle the client's request, and the load balancer forwards the request to the chosen server.
4. Response Handling:
    a. Once the server processes the client's request, the load balancer receives the response from the server.
    b. The load balancer efficiently forwards the server's response back to the originating client.

TCP Connection Management:

1. Maintaining Server Connection:
    a. For TCP requests, the load balancer aims to maintain the connection between the client and the selected server throughout the entire TCP session.
    b. To achieve this, the load balancer establishes and manages a mapping for each client:port to the corresponding server's IP. This mapping ensures that the server

consistently handles requests from the same client during tot the appropriate server to maintain the active TCP connection.

Dynamic Load Balancing:

2. Tracking Active Connections:
    a. In the context of dynamic load balancing, the load balancer keeps track of the number of active connections each server is handling. This information helps distribute incoming requests evenly across the available servers.
3. Monitoring Response Times:
    a. During the continuous ping probes, the load balancer records response times from each server. This data contributes to the decision-making process in selecting the fastest and most responsive server for handling client requests.
    b. By following this comprehensive approach, the load balancer optimizes the distribution of client requests among servers, ensuring efficient load distribution and responsive server selection based on real-time health and performance metrics.

Instructions

Installation and Setup:

1. Start Mininet VM
    - Ensure that the VM is connected to the internet
2. Install prerequisites
    a. Update package list

```
sudo apt-get update
```

b. Install curl:

```
sudo apt-get install curl
```

c. Fix this line of code in ./pox/lib/packet/packet_utils.py

Update the line around 105 in check_sum function, to fix a type conversion error:

```
start += struct.unpack('H', bytes([data[-1], 0]))[0]
```

3. Repository setup
    a. Clone repository

```
git clone https://github.com/Lordpritish/LoadBalancer.git
```

    b. Copy all files (except mytop.py) from the repository to the POX extension folder. Copy mytop.py from the repository to the Mininet examples folder

```
chmod +x copy_files.sh
```

```
./copy_files.sh
```

Running the Load Balancer

1. Load Balancer Controller

The POX controller implements both static and dynamic load balancing algorithms with support for four strategies:

- Random Balancer (RANDOM): Random server selection.
- Round Robin (ROUND_ROBIN): Circular sequence server selection.
- Weighted Round Robin (WEIGHTED_ROUND_ROBIN): Server selection based on assigned weights.
- Least Response Time (LEAST_RESPONSE_TIME): Server selection based on response time and open connections

2. Starting the Controller
    a. Navigate to the POX directory

```
cd ./pox
```

    b. Terminate any existing controllers

```
sudo pox.py killall controller
```

    c. Run the load balancer controller:

```
./pox.py log.level --DEBUG LoadBalancer --ip=10.0.1.1
--servers=10.0.0.1,10.0.0.2,10.0.0.3,10.0.0.4
```

```
./pox.py log.level --DEBUG LoadBalancer --ip=10.0.1.1
--servers=10.0.0.1,10.0.0.2,10.0.0.3,10.0.0.4 --alg=1
```

```
./pox.py log.level --DEBUG LoadBalancer --ip=10.0.1.1
--servers=10.0.0.1,10.0.0.2,10.0.0.3,10.0.0.4 --alg=3
--weights=1,2,3,4
```

```
./pox.py log.level --DEBUG LoadBalancer --ip=10.0.1.1
--servers=10.0.0.1,10.0.0.2,10.0.0.3,10.0.0.4 --alg=4
```

3. Mininet Example
    a. Navigate to the Mininet examples directory

```
cd ~/mininet/examples/
```

    b. Clear existing Mininet configurations

```
sudo mn -c
```

    c. Prepare for xterm usage

```
sudo cp ~/.Xauthority ~root/
```

d. Run mininet topology

```
sudo python ./mytop.py
```

4. Testing the Load Balancer
    a. Use ping to send 10 packets from each client to the load balancer IP address, for example:

    ```
    h1 ping -c 10 10.0.1.1
    ```

5. Generating Traffic Plots
    a. After terminating Load Balancer, it generates a req_count.txt file. To generate summary plots, run:

    ```
    python3 <Path-to-UIGenerator.py> <Path-to_req_count.txt>
    ```

Implementation Details

The project leverages Software-Defined Networking (SDN) principles, particularly using the POX controller, to implement a dynamic load balancing system. The system comprises several Python scripts and modules, each serving distinct roles:

1. LoadBalancer.py: The core component that integrates different load balancing algorithms to manage network traffic across multiple servers.
2. Algorithm Modules (LoadBalancingAlgthm.py): Contains the implementation of various load balancing strategies such as Round Robin, Weighted Round Robin, and Least Response Time.
3. Network Handlers (ArpHandler.py, PingHandler.py): Manage network-level operations like ARP handling and server health checks via ICMP pings.
4. Flow Rule Management (FlowRule.py): Utilizes the OpenFlow protocol to define rules for routing network packets.

5. UI and Visualization Tools (LBTKinter.py, UIGenerator.py): Provides a graphical user interface and tools for visualizing traffic flow and load balancer performance.
6. Request Logging (RequestLogWriter.py): Captures and logs details of network requests for analysis.
7. Mininet Scripts (mytop.py, parallelRequest.py): Facilitates the creation of a virtual network environment and simulates client requests for testing purposes.
8. Testing Scripts (client.py, server.py): Essential for system validation, client.py emulates clients, sending requests to the load balancer, while server.py simulates backend servers, handling routed requests. Together, they test the load balancer's efficiency and request distribution in a controlled environment.

LoadBalancer.py

Key Components and Functionalities

1. Module Imports and Global Variables
    - Core Libraries: Uses pox.core, pox.lib.addresses, pox.openflow.libopenflow_01, and pox.lib.packet.
    - Custom Classes: Imports FlowRuleManager, ARPHandler, PingHandler, and various load balancing algorithm classes from LoadBalancingAlgthm.py.
    - Algorithm Identifiers: Defines constants like RANDOM, ROUND_ROBIN, etc., for easy selection of load balancing algorithms.

2. Class Definition: SimpleLoadBalancer

- Initialization: Configures the load balancer with the specified IP, MAC address, servers, and selected balancing algorithm.
- Listeners: Adds listeners to OpenFlow events for handling network packets.
- Server Management: Maintains a list of servers and corresponding MAC-to-port mappings.
- Connection Tracking: Keeps track of ongoing client connections and their associated server.
- Ping Probing: Implements functionality for periodic server health checks using ping probes.

3. Load Balancing Algorithm Integration
    - Dynamic Selection: Allows the selection of different algorithms (e.g., Random, Round Robin) during runtime.
    - Algorithm Objects: Instantiates the chosen load balancing algorithm and uses it for server selection and traffic distribution.

4. Packet Handling
    - ARP Handling: Manages ARP requests and responses to ensure correct IP-to-MAC resolution in the network.
    - IP Packet Routing: Directs incoming IP packets to the appropriate server based on the load balancing algorithm.
    - TCP Connection Management: Tracks TCP connections' states (e.g., new, in progress, ended) and updates server load accordingly.
    - ICMP Handling: Processes ICMP packets, particularly for ping responses, to monitor server health and response times.

5. Flow Rule Management

- OpenFlow Integration: Utilizes OpenFlow protocol for defining flow rules that direct traffic to backend servers.
- Dynamic Rule Updates: Adjusts flow rules based on changes in server availability and load.

ArpHandler.py

Overview

ArpHandler.py is responsible for managing Address Resolution Protocol (ARP) related tasks within the load balancer. It plays a vital role in ensuring correct IP-to-MAC address mappings, which are crucial for proper network packet routing.

Key Features

- Initialization: Configured with the load balancer's MAC and IP addresses, and a broadcast MAC address.
- Proxied ARP Reply: Sends ARP replies on behalf of other network entities, crucial for masquerading the load balancer as the destination host.
- Proxied ARP Request: Generates ARP requests to discover MAC addresses for specific IP addresses, maintaining accurate IP-to-MAC mapping.

FlowRule.py

Overview

FlowRule.py is tasked with managing the flow rules in the SDN environment. This component interacts directly with the OpenFlow protocol, defining how packets should be handled and routed through the network.

Key Features

- Packet Out Message Construction: A method for constructing and sending packet_out messages. This is used to manually forward packets through the network, particularly useful in scenarios where the load balancer needs to directly control packet routing.

LoadBalancingAlgthm.py

1. Common Base Class: LoadBalancer (Super Class)
    - Functionality: Provides a template for load balancing algorithms with methods to add, delete, and retrieve servers.
    - Server Management: Maintains a list of live servers and their count.
2. LeastResponseTimeBalancer (Inherits LoadBalancer)
    - Algorithm: Prioritizes servers with the lowest active connections and shortest average response time.
    - Implementation:
        a. Manages server response times dict (maps server_Ip -> response time) and active connection counts dict (maps server_Ip -> # of connections) .
        b. Uses a round-robin approach when multiple servers have similar metrics.
        c. Dynamically updates server response times and connection counts.
    - Server Selection:
        a. Find the servers with the lowest active connections server from active connections dictionary.
        b. If there are multiple servers with the lowest active connections, find the server/s with the shortest average response time. Following are some of the cases to note:
            - Case1: If there are multiple servers with the same shortest average response time, then apply the round-robin method and assign the new request to the server that has its turn

        - Case2: If the server is exactly one, assign the new request to this server.
    c. If the server is exactly one, assign the new request to this server.Chooses a server randomly from the list of live servers
3. RandomBalancer (Inherits LoadBalancer)
    - Functionality: Selects a server randomly from the list of live servers.
    - Server Removal: Implements logic to remove a server from the list when needed.
    - Server Selection: Choose a server randomly from the list of live servers
4. RoundRobinBalancer (Inherits LoadBalancer)
    - Algorithm: Distributes requests in a circular order among servers.
    - Server Index Tracking: Maintains an order list of live servers and a current index to keep track of the next server in the sequence.
    - Server Selection: Choose the server that's next on the list order of live servers
    - Server Removal: Adjusts the index and server count when a server is removed.
5. WeightedRoundRobinBalancer (Inherits RoundRobinBalancer)
    - Weighted Distribution: Assigns weights to servers, giving preference to servers with higher weights in the round-robin sequence.
    - Dynamic Weight Adjustment: Adjusts weights in real-time to ensure fair distribution according to the set weights.
    - Server Selection: Choose the server based on their weights, and while giving more preference to servers who have higher weights.
    -

PingHandler.py

Purpose: Handles ICMP ping requests to monitor servers health and availability.

<u>Key Features:</u>

- Ping Request Generation: Creates and sends ICMP echo (ping) requests to specific hosts, using the load balancer's MAC and IP addresses.
- Network Packet Construction: Builds Ethernet and IPv4 packets to encapsulate ICMP messages, facilitating server health checks.

Client.py

Functionality: Simulates client requests to the load balancer for testing and validation purposes.

Operation:

- Request Generation: Sends HTTP requests to the load balancer's IP address.
- Test Control: Allows configuration of the number of requests and the delay between each request, facilitating various test scenarios

Server.py

Purpose: Emulates backend server behavior in a load-balanced environment.

implementation:

- HTTP Server: Utilizes Python's SimpleHTTPServer and SocketServer to create a basic HTTP server responding to GET requests.
- Response Customization: Customizes server responses to include the responding server's identification, aiding in validating the load balancer's distribution mechanism.

mytop.py:

Functionality: Defines and initiates a custom network topology in Mininet for testing the load balancer.

Components:

- Controller and Switches: Configures a remote controller and Open vSwitch switches.
- Hosts Addition: Dynamically adds multiple hosts (servers and clients) to the network.
- HTTP Server Initialization: Starts simple HTTP servers on select hosts for testing.
- Network Initialization and CLI: Builds the network, starts controllers and switches, and provides a CLI for interaction.

RequestLogWriter.py:

Functionality: Logs requests processed by the load balancer for analysis.

Implementations:

- In LoadBalancer.py there is a listener function that gets called everytime there is packet through the loadbalancer named `handle_ip_packet()`. In this function, I would listen to the packet and check for the appropriate flags to detect the start of the connection and the end of the connection. (SYN for start, FIN and RST for done)
- For every start or end of the connection, I write the server the load balancer is redirecting the request to, the time, and whether it is start/end.
- When terminating the load balancer, we also terminate the write process to the log file

```
260                 #chekc if a ping reequest
261                 if ip_packet.protocol != pkt.ipv4.TCP_PROTOCOL:
262                     self.req_log_writer.write_request(str(server_ip), "start")
263
264                 #  if ip, we know the connction start with SYN and finished with FIN
265                 tcp_found = packet.find('tcp')
266                 if tcp_found:
267                     if tcp_found.SYN:
268                         self.req_log_writer.write_request(str(server_ip), "start")
269                     elif tcp_found.FIN:
270                         self.req_log_writer.write_request(str(server_ip), "done")
```

UIGenerator.py

Functionality: Generate summary charts of the load balancer and servers' performance from req_count.txt

Components:

- Number of requests redirected to each server:
    - One with a bar graph
    - One with a chart that shows the total # of requests through the timeline
- Bar graph showing the average connection time for each server
    - This represents the latency of the server in most cases (the longer the connection time, the slower the response)
- Chart showing the number of active connections through the timeline

Implementations:

- First, it would create 2 dictionaries from req_count.txt, one that maps server to the list of time that the connection has started to that server and the other one

maps server to the list of time that the connection has ended (`server_to_start_connection` and `server_to_start_connection` respectively)

- From these two dictionaries, we create plot each of the charts and graphs using Matplotlib library
- Had different proto types for the graphs and charts but final version of the charts has clear and concise looks and will help users identify the performance of the load balancer at a glance.

```
if __name__ == '__main__':
    if (len(sys.argv) < 2):
        std.error("Please enter the log text file")

    f = open(sys.argv[1], "r")
    server_to_start_connection, server_to_done_connection, lastTime = construct_lists_from_file(f)
    f.close()
    fig = plt.figure(figsize= (10, 5))
    server_to_num_request(server_to_start_connection)
    plot_req_over_time_count(deepcopy(server_to_start_connection), lastTime)
    plot_net_req_count(deepcopy(server_to_start_connection), deepcopy(server_to_done_connection), lastTime)
    plot_average_response_time(deepcopy(server_to_start_connection), deepcopy(server_to_done_connection))
```

parallelRequest.py:

Purpose: Simulates concurrent requests to the load balancer for stress testing.

Methodology:

- HTTP Requests: Uses the requests library to send HTTP requests to the load balancer.
- Multiprocessing: (Commented out) Provides an option for parallel request execution using Python's multiprocessing module.
- Session Handling: Utilizes requests.Session for managing HTTP sessions.

Analysis

The following are sample outputs from running the load balancer

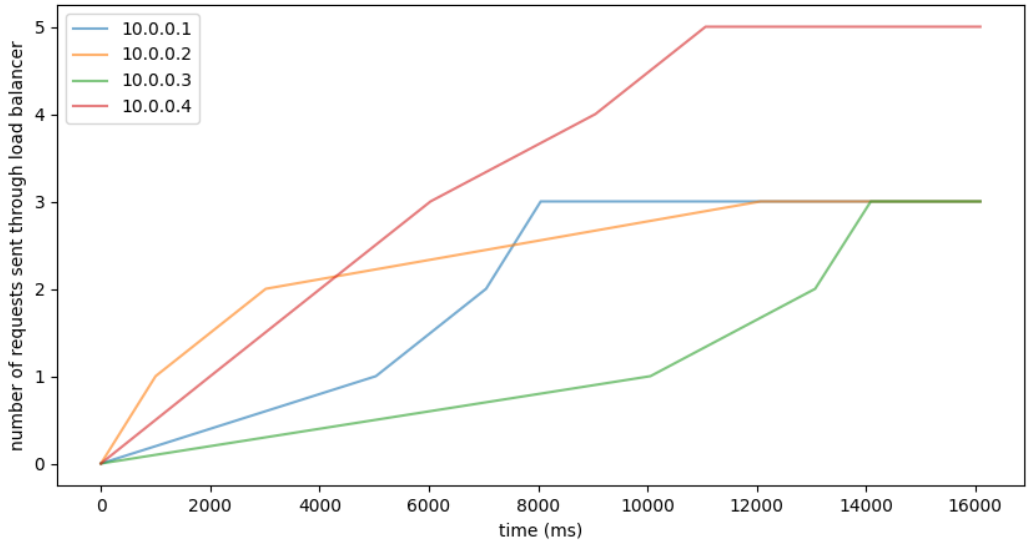

Figure 1: number of requests through load balancer

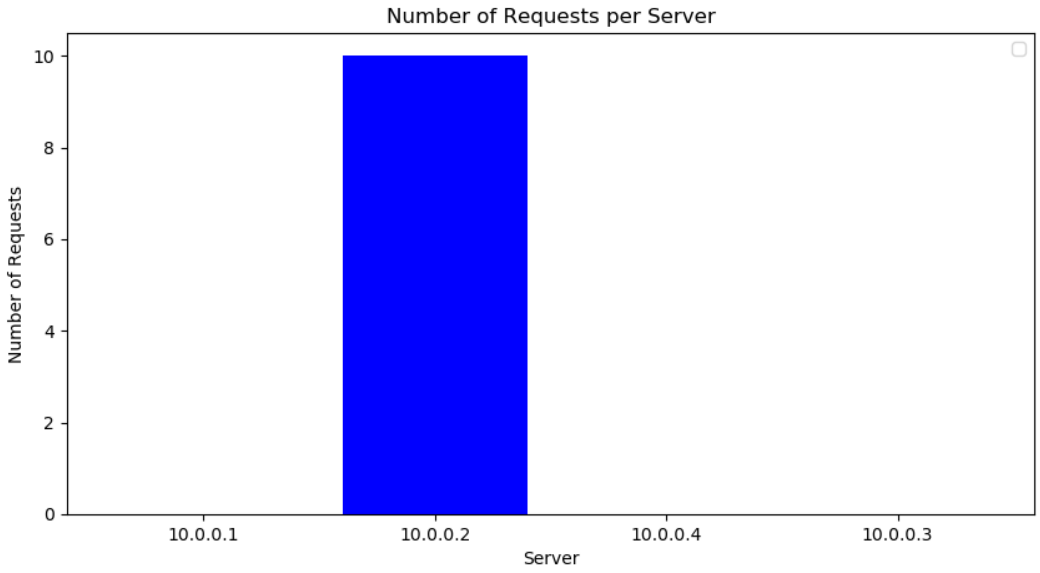

Figure 2: number of requests

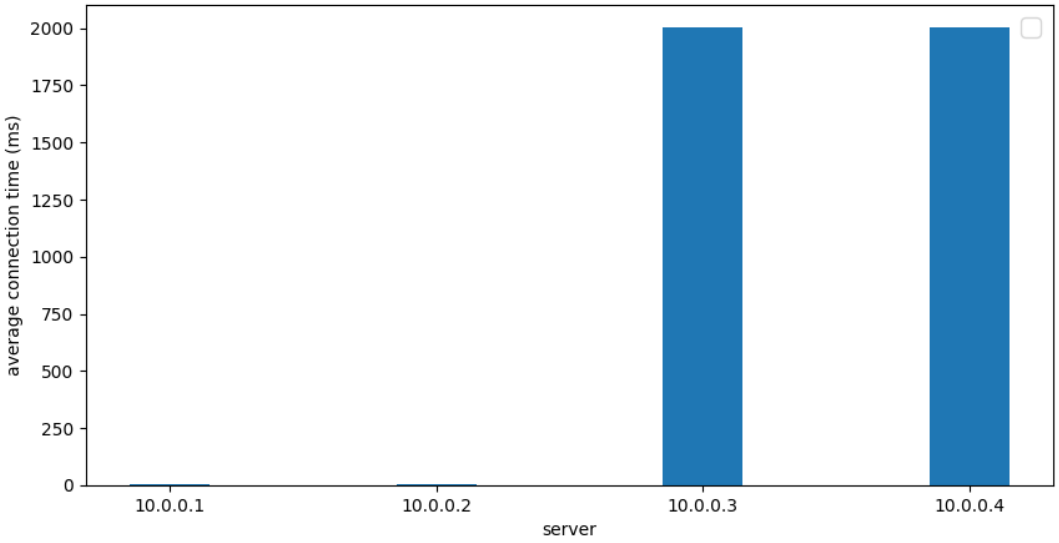

Figure 3: average connection time

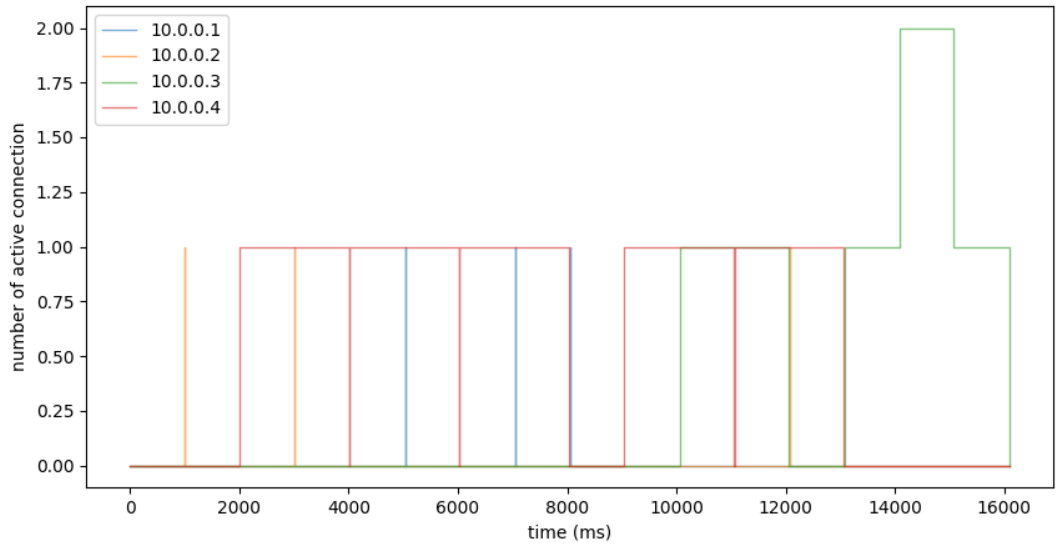

Figure 4: number of active connection

In Figure 1, it can be observed how requests are being handled by individual servers. The load balancer appears to be distributing requests across all servers. The different slopes for each line suggests that the load balancer is using a dynamic load balancing

algorithm. In Figure 2, the bar chart displays the total number of requests handled by each server. The distribution of requests is not uniform, which is due to the dynamic nature of the algorithms adjusting to server response times. For example, it may have been assigned a higher weight in a weighted round robin algorithm. The third graph shows the average connection time for each server, the high connection time for the two servers suggests that the load balancer is using a dynamic algorithm. Figure 4 shows the number of active connections over time for each server. The sudden increase in active connections for server 10.0.0.3 indicates that the load balancer temporarily favored this server, which demonstrates the adaptability of the load balancer to reallocate requests to optimize the network conditions.

Concluding remarks

This project effectively demonstrates the application of SDN principles to create a dynamic load balancing system. We have successfully explored the potential of SDN to improve network performance through intelligent load balancing and gained valuable insights into the application development

With a variety of load balancing algorithms such as round robin, weighted round robin, and least response time, we have highlighted SDN's flexibility in managing network traffic and its capacity to enhance overall network performance. The dynamic nature of the algorithm selection offers the flexibility to adapt to various conditions and optimize traffic flow effectively

The addition of a user-friendly web interface bridged the gap between complex network management tasks and end-user interaction, making advanced load balancing techniques more accessible and manageable. This interface serves as a useful tool for monitoring the impact of different load balancing strategies, allowing users to make informed decisions to enhance network performance.

This project has laid a foundation for further exploration into more advanced SDN features, such as the possibility of integrating machine learning for predictive load balancing, or the exploration into more complex network environments.